Programming Perl

Fourth Edition

Tom Christiansen, brian d foy,
Larry Wall and Jon Orwant

O'REILLY®

Программирование на Perl

Четвертое издание

Том Кристиансен, брайан д фой, Ларри Уолл и Джон Орвант



Санкт-Петербург — Москва
2014

Том Кристиансен, брайан д фой,
Ларри Уолл и Джон Орвант

Программирование на Perl, 4-е издание

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научные редакторы	*М. Зислис, А. Киселев*
Редактор	*М. Зислис*
Верстка	*Д. Орлова*
Корректоры	*Т. Иванкова, В. Логунова*

Кристиансен Т., д фой б., Уолл Л., Орвант Дж.

Программирование на Perl, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2014. – 1048 с., ил.

ISBN 978-5-93286-214-8

Четвертое издание «Программирование на Perl» ждали в России и программисты, и системные администраторы. И вот обновление описания языка Perl, продолжавшего активно развиваться в течение последних пяти лет, перед вами. На этот раз в «Кэмэл» обсуждается текущая версия Perl 5.14 и дается обзор некоторых особенностей готовящейся к выходу версии Perl 5.16.

Все большую значимость в обработке текстов приобретает Юникод, а Perl предлагает лучшую и самую безболезненную поддержку этого стандарта, тесно интегрируя Юникод во все сферы, в том числе в такой популярный механизм языка Perl, как регулярные выражения. И Юникоду, и регулярным выражениям уделено много внимания в книге.

Четвертое издание охватывает такие важные особенности языка Perl, как новые ключевые слова и синтаксические конструкции, уровни ввода/вывода и кодировки, новые escape-последовательности, поддержка стандарта Unicode 6.0, групповые графемы и свойства символов Юникода, именованные сохраняющие группы в регулярных выражениях, рекурсивные и грамматические шаблоны, расширенный обзор архива CPAN и современные передовые приемы программирования. Материал иллюстрируется множеством интересных примеров. И если вы ищете справочник по языку Perl, то он перед вами.

ISBN 978-5-93286-214-8
ISBN 978-0-596-00492-7 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 12.11.2013. Формат 70×100 1/16.
Печать офсетная. Объем 67,5 печ. л.

Оглавление

Предисловие

В погоне за счастьем

Perl – язык, с помощью которого вы сделаете свою работу.

Конечно, если эта работа – программирование, то теоретически ее можно сделать с помощью любого «полного» компьютерного языка. Но опыт показывает, что компьютерные языки различаются не столько *возможностью* что-либо сделать, сколько *легкостью*, с которой это достигается. На одном полюсе находятся так называемые языки четвертого поколения, с помощью которых можно легко делать одни вещи и почти невозможно другие. На другом полюсе – так называемые языки с промышленными возможностями (industrial-strength languages), посредством которых одинаково трудно делать почти все.

Perl не таков. Если сказать кратко, по задумке его создателей *на этом языке легко решать простые задачи, сохраняя возможность решать и сложные.*

Что это за «простые задачи», которые должны решаться легко? Разумеется, те, которые мы решаем изо дня в день. Нам нужен язык, с помощью которого легко работать с числами и текстом, файлами и каталогами, компьютерами и сетями, а в особенности – с программами. Он должен позволять легко запускать внешние программы и просматривать результаты их работы в поисках интересных данных. Он должен позволять легко отправлять эти интересные данные другим программам, способным обрабатывать их особым образом. Он должен также позволять нам легко разрабатывать собственные программы, изменять их и производить отладку. И конечно, наши собственные программы должны легко компилироваться и запускаться, а также быть переносимыми на любую современную операционную систему.

Все это, а также многое другое делает Perl.

Первоначально разработанный как интегрирующий язык для UNIX, Perl давно распространился на большинство других операционных систем. Поскольку Perl выполняется почти везде, он является одной из наиболее переносимых сред программирования, существующих сегодня. Чтобы писать переносимые программы на C или C++, необходимо расставить все эти странные пометки `#ifdef` для каждой операционной системы. Для обеспечения переносимости программ на Java нужно разбираться в индивидуальных особенностях всех реализаций этой платформы. Для создания переносимых сценариев командной оболочки нужно помнить синтаксис всех команд для каждой версии операционной системы и пытаться найти общий знаменатель, благодаря которому они, как можно надеяться,

будут работать всюду. А чтобы создавать переносимые программы на Visual Basic, потребуется дать более гибкое определение понятия «переносимость». :-)

Perl позволяет нам счастливо избежать таких проблем, сохраняя при этом многие преимущества других языков и добавляя собственные чудеса. У этих чудес много источников: практичность набора функций Perl, изобретательность сообщества Perl и неистребимый энтузиазм движения open source в целом. Однако в значительной мере чудеса обусловлены гибридной природой Perl. У Perl смешанное происхождение, и многообразие средств в этом языке всегда считалось плюсом, а не слабостью. Perl – это язык, говорящий: «Дайте мне ваших усталых, ваших бедных»[1]. Если вы чувствуете себя словно в теснящейся толпе и стремитесь «дышать свободно», то Perl – для вас.

Perl охватывает различные культуры. Его взрывное распространение в значительной мере питалось стремлением *бывших* системных UNIX-программистов взять с собой как можно больше из «старого мира». Для них Perl является переносимой квинтэссенцией культуры UNIX, оазисом в пустыне «невозможности перейти из одного места в другое». Существует, однако, и движение в обратном направлении: веб-дизайнеры, работающие в Windows, с удовольствием обнаруживают возможность запускать свои Perl-программы на UNIX-сервере своей компании без доработки.

Хотя Perl особенно популярен среди системных программистов и веб-разработчиков, это связано лишь с тем, что они первыми его открыли; аудитория Perl значительно шире. Получив при создании скромный статус языка обработки текста, Perl развился в сложный язык программирования общего назначения с богатой средой разработки программ, укомплектованной отладчиками, профилировщиками, компоновщиками, компиляторами, библиотеками, редакторами с подсветкой синтаксиса и другими атрибутами «настоящего» языка программирования – если они вам требуются. Но все они относятся к поддержке возможностей решения сложных задач, с чем справляются многие другие языки. Уникальность Perl в том, что он никогда не отступал от идеи легких решений для простых задач.

Поскольку Perl является одновременно мощным и доступным средством, он постоянно используется во всех мыслимых сферах – от аэрокосмической техники до молекулярной биологии, от математики до лингвистики, от графики до обработки документов, от управления базами данных до сетевого администрирования. Perl используется теми, кому позарез нужно быстро проанализировать или преобразовать большие объемы данных, будь то последовательность генов ДНК, набор веб-страниц или контракты на поставку свинины.

Существует много слагаемых успеха этого языка. Perl как открытый проект стал успешным еще до того, как движение open source получило свое название. Perl свободно распространяется, и так будет всегда. Каждый может работать с Perl так, как сочтет удобным, и на основе очень либеральной политики лицензирования. Если вы занимаетесь коммерческой деятельностью и хотите воспользоваться Perl, можете приступать. Язык Perl разрешается встраивать в коммерческие приложения бесплатно и без ограничений. А для тех, у кого возникнет проблема,

---

[1] «Give me your tired, your poor, Your huddled masses yearning to breathe free...» – эти слова являются частью текста стихотворения, выбитого на табличке, которая находится в помещении внутри пьедестала Статуи Свободы. См. фото таблички по адресу *http://en.wikipedia.org/wiki/The_New_Colossus*. – *Прим. ред.*

которую сообщество Perl не сможет решить, существует безотказная страховка – сам исходный код. Сообщество Perl не занимается продажей своих профессиональных тайн под видом «обновлений». Сообщество Perl никогда не «выйдет из дела» и не оставит вас с брошенным на произвол судьбы продуктом.

Безусловно, популярности Perl способствует его бесплатное распространение. Но этого недостаточно для объяснения феномена Perl, поскольку большой успех приходит далеко не ко всем бесплатно распространяемым пакетам. Дело не в том, что он бесплатен; он доставляет удовольствие. Люди чувствуют желание творить на Perl, поскольку он дает свободу самовыражения: можно выбирать между целями оптимизации – скоростью работы компьютера или скоростью программирования, между многословием и выразительностью, между «читабельностью» и простотой поддержки или повторного использования, или переносимости, или простотой изучения, или поучительностью. Можно оптимизировать даже непонятность, если принять участие в конкурсе на самую непонятную программу – Obfuscated Perl Contest.

Perl способен предоставить все эти степени свободы, поскольку является языком с раздвоением личности. Это одновременно и очень простой, и очень богатый язык. Perl заимствует лучшие идеи практически отовсюду и объединяет их в простую логическую систему. Для тех, кому он просто нравится, Perl – это *Practical Extraction and Report Language* (практический язык извлечения данных и создания отчетов). Для тех, кто любит его, Perl – это *Pathologically Eclectic Rubbish Lister* (паталогически эклектичный язык для распечатки чепухи). А минималистам Perl кажется проявлением бесцельной избыточности. Но это хорошо. Редукционисты должны существовать (в основном среди физиков). Редукционисты стремятся разъять целое на части. Мы, все остальные, просто пытаемся собрать целое из частей.

Во многих отношениях Perl – простой язык. Не требуется знать множество особых заклинаний, чтобы скомпилировать программу на Perl – ее можно просто выполнить как пакетный файл или сценарий оболочки. Типы и структуры Perl просты в использовании и понимании. Perl не налагает свои произвольные ограничения на данные – строки и массивы могут быть сколь угодно велики, лишь бы хватило оперативной памяти, и их организация позволяет им легко увеличиваться по мере надобности. Perl не требует изучения новых синтаксиса и семантики, в значительной мере заимствуя их из других языков, с которыми вы можете быть знакомы (например, C, *awk*, BASIC, Python, английский и греческий). На практике почти любой программист сможет прочесть хорошо написанный код на Perl и составить себе представление о том, что он делает.

Очень важно, что нет необходимости изучать Perl полностью, чтобы начать писать полезные программы. Изучение Perl можно начать с «тонкого конца». Вы можете программировать на Perl версии «Детский Лепет», и мы обещаем не смеяться над этим. Точнее, мы обещаем смеяться не более чем над первыми попытками ребенка творчески подходить к миру. Многие идеи Perl заимствованы из естественного языка, и одна из лучших его черт состоит в том, что он позволяет использовать лишь подмножество языка, если его достаточно, чтобы передать мысль. В культуре Perl приемлема любая степень владения языком. Полицию по охране языка мы к вам не пришлем. Сценарий Perl будет «правильным», если выполнит задачу прежде, чем начальник вас уволит.

Будучи во многих отношениях простым, Perl является и богатым языком, в котором можно долго совершенствоваться. Это расплата за возможность решать сложные задачи. Хотя понадобится некоторое время на освоение всех средств Perl, вы будете рады иметь в своем распоряжении расширенные возможности, когда они вдруг понадобятся.

Благодаря своему происхождению Perl был богатым языком уже тогда, когда считался «просто» языком преобразования данных, предназначенным для ориентирования в файлах, просмотра больших объемов текста, создания и получения динамических данных и вывода легко форматируемых отчетов, основанных на этих данных. Но в какой-то момент начался расцвет Perl. Он стал также и языком для работы с файловой системой, управления процессами, администрирования баз данных, программирования в архитектуре клиент-сервер, создания безопасных программ, управления данными в Сети и даже для объектно-ориентированного и функционального программирования. Эти возможности не были просто механически присоединены к Perl – каждая новая синергически работает с остальными, поскольку с самого начала Perl проектировался как интегрирующий язык.

Но Perl умеет объединять в единое целое не только собственные функции. Он создавался как модульный, расширяемый язык. Perl позволяет быстро проектировать, программировать, отлаживать и разворачивать приложения, а также без труда расширять функциональные возможности этих приложений при необходимости. Perl можно встраивать в другие языки, а другие языки можно встраивать в Perl. С помощью механизма импорта модулей можно использовать эти внешние определения, как если бы они были встроенными функциями Perl. Объектно-ориентированные внешние библиотеки сохраняют свою объектную ориентированность в Perl.

Perl помогает разработчику и в других отношениях. В отличие от строго интерпретируемых языков, таких как командные файлы и сценарии оболочки, которые компилируют и выполняют лишь одну команду за раз, Perl сначала быстро компилирует всю программу в промежуточный формат. Подобно любому другому компилятору, он осуществляет различного вида оптимизации и мгновенно реагирует на любые ошибки – от синтаксических и семантических до неудачи при связывании с библиотеками. Когда компилирующий интерфейс Perl удовлетворен вашей программой, он передает промежуточный код на выполнение интерпретатору (либо какому-либо из нескольких модулей генераторов, способных создавать текст на C или байт-код). Все это выглядит сложным, однако компилятор и интерпретатор работают весьма эффективно, и обычный цикл компиляции-прогона-исправления занимает считанные секунды. В совокупности с мощной поддержкой амортизации отказов столь короткий цикл делает Perl языком, на котором действительно возможно быстрое прототипирование. Позже, по ходу совершенствования программы, вы сможете повысить требования к себе и программировать больше за счет дисциплины, чем за счет интуиции. Perl и в этом окажет содействие, если его вежливо об этом попросить.

Perl также способствует созданию более защищенных программ. Помимо всех обычных интерфейсов защиты, предоставляемых другими языками, Perl защищает от случайных ошибок в системе безопасности посредством уникального механизма трассировки данных, автоматически определяющего данные, которые поступили из ненадежного источника, и предотвращающего выполнение опасных операций. Наконец, Perl позволяет создавать специальные защищенные отсеки,

в которых можно безопасно выполнять код сомнительного происхождения с ограничением опасных операций.

Парадоксально, но самая большая помощь, которую Perl может оказать программисту, связана не столько с языком Perl, сколько с людьми, которые с ним работают. Скажем откровенно, сообщество Perl составляют люди, которые более чем кто-либо другой готовы прийти на помощь. Если считать, что в движении Perl есть что-то благочестивое, то именно это и есть его основная ценность. Ларри хотел, чтобы сообщество было чем-то вроде рая, и в целом его желание пока осуществляется. Внесите и свой вклад, чтобы оно таким и оставалось.

Изучаете ли вы Perl ради спасения мира, или из любопытства, или по приказу вашего начальника – в любом случае этот учебник позволит освоить как основы, так и сложные вопросы. И хотя мы не намереваемся учить вас программированию, проницательный читатель что-то приобретет как от искусства, так и от науки программирования. Мы рекомендуем вам развивать в себе три великие добродетели программиста: *лень*, *нетерпеливость* и *высокомерие* (*laziness*, *impatience*, *hubris*). Мы надеемся, что, читая эту книгу, вы найдете ее местами довольно занимательной (а местами – крайне занимательной). Если этого окажется недостаточно, чтобы вы не заснули, постоянно напоминайте себе, что изучение Perl повысит ценность вашего резюме. Так что читайте дальше.

Что нового в этом издании

Проще сказать, что старого! Прошло достаточно много времени с момента выхода предыдущего издания. В свое оправдание мы можем лишь сказать, что у нас была на то пара причин, но теперь все в порядке.

Третье издание вышло в середине 2000 года, как раз когда вышла версия Perl 5.6. Когда мы пишем эти строки 12 лет спустя, к выходу готовится версия Perl 5.16. Много воды утекло за эти годы: вышло несколько новых версий Perl 5 и случилось маленькое событие, которое мы называем Perl 6. Однако эта шестерка обманчива. В действительности, Perl 6 – это «младший брат» Perl 5, а не важное обновление Perl 5, как можно было бы заключить из номера версии. Но в этой книге не рассказывается об этом другом языке. Она все еще посвящена Perl 5 – версии, которой вполне успешно пользуется большинство людей во всем мире (и даже парни из проекта Perl 6!).[1]

Чтобы рассказать, что нового в этой книге, придется рассказать, что нового в Perl. Это издание – не просто «косметический ремонт», призванный повысить продажи книги. Это долгожданное обновление описания языка, продолжавшего активно развиваться в течение последних пяти лет. Мы не будем перечислять все изменения (при необходимости обратитесь к страницам *perldelta*), но есть кое-что, о чем нам хотелось бы рассказать отдельно.

В Perl 5 мы начали добавлять новые функциональные возможности, одновременно создавая средства защиты старых программ от новых инструкций. Например,

[1] Поскольку мы ленивы, а вы уже знаете, что эта книга посвящена Perl 5, следует упомянуть, что мы не всегда будем точно указывать номер версии «Perl v5.n» – в оставшейся части книги, если вы увидите лишь номер версии, начинающийся с «v5», просто знайте, что речь идет об этой версии Perl.

мы наконец смягчились в отношении частых просьб реализовать инструкцию, подобную инструкции switch. Однако, как это принято в мире Perl, мы сделали ее лучше и удобнее, предоставив вам более полный контроль над тем, что вы делаете. Мы назвали ее given-when, но эта инструкция будет доступна, только если вы явно попросите об этом. Любая из следующих директив включает доступ к этой новой возможности:

```
use v5.10;
use feature qw(switch);
use feature qw(:5.10);
```

а включив ее, вы получаете «заряженный» оператор switch:

```
given ($item) {
    when (/a/)   { say "Matched an a" }
    when (/bee/) { say "Matched a bee" }
}
```

В главе 4 вы познакомитесь поближе с этой и другими новыми особенностями, так как там их обсуждение более уместно.

Даже при том что поддержка стандарта Юникод (Unicode) существует в Perl начиная с версии v5.6, она была значительно улучшена в последних версиях. В частности, это касается более полной, чем в других языках программирования на данный момент, поддержки Юникода в регулярных выражениях. Благодаря постоянному улучшению поддержки этого стандарта Perl иной раз используется даже для испытаний будущих наработок «Консорциума Юникода». В предыдущем издании этой книги весь материал, посвященный Юникоду, уместился в единственной главе, а в этом издании обсуждение этой темы встречается везде, где это уместно.

Стали еще лучше регулярные выражения – одна из особенностей, устойчиво ассоциирующихся у программистов именно с языком Perl. Другие языки заимствовали язык шаблонов из Perl и дали ему название Perl Compatible Regular Expressions (регулярные выражения, совместимые с Perl), но при этом добавили некоторые свои особенности. Мы, в свою очередь, заимствовали некоторые из этих особенностей, продолжая традицию вбирать в Perl все самое лучшее отовсюду. Вы также познакомитесь с мощными новыми функциями для работы с Юникодом в шаблонах регулярных выражений.

Потоки выполнения (threads) также претерпели значительные изменения. Perl поддерживает две модели многопоточного выполнения: одну мы назвали 5005threads (по номеру версии, в которой она была добавлена), а другая – потоки интерпретатора. Начиная с версии v5.10 поддерживаются только потоки интерпретатора. Однако по различным причинам мы решили не включать обсуждение этой темы в книгу, а больше внимания уделить другим особенностям. Если у вас появится желание изучить потоки выполнения, обратитесь к странице *perlthrtut* справочного руководства[1], которая содержит практически все, что мы могли бы поместить в главу, описывающую потоки выполнения. Возможно, в будущем мы добавим эту главу в качестве бесплатного приложения.

---

[1] Доступ к соответствующим страницам можно получить при помощи команды *perldoc perlthrtut* или по адресу *http://perldoc.perl.org/perlthrtut.html*. – *Прим. ред.*

С течением времени одни особенности появлялись, другие исчезали. Некоторые экспериментальные особенности оказывались неудачными, и мы заменяли их другими экспериментальными особенностями. Так были убраны и забыты псевдохеши[1]. Если вы не знаете, что это такое, не переживайте об этом и не ищите их в этом издании.

И еще: с момента последнего обновления этой книги произошла крупная революция (или две) в практике программирования на Perl, как и в культуре тестирования. Архив CPAN (Comprehensive Perl Archive Network – обширный сетевой архив ресурсов для Perl) продолжает свой экспоненциальный рост, что делает его «убойной особенностью» Perl. Хотя эта книга и не об архиве CPAN, мы все же будем рассказывать о некоторых модулях из него, когда это потребуется. Не пытайтесь реализовать все на голом Perl, без использования дополнительных модулей.

Мы исключили из этого издания две главы: список модулей стандартной библиотеки и список диагностических сообщений (главы 32 и 33 в предыдущем издании). Обе они устареют еще до того, как эта книга попадет на вашу книжную полку. Мы расскажем вам, как самостоятельно получить этот список. Что касается диагностических сообщений, их можно найти на странице *perldiag* справочного руководства[2] или включить вывод подробных предупреждений прагмой `diagnostics`.

Часть I «Общий обзор»

Начать всегда труднее всего. В этой части базовые идеи Perl излагаются в неформальном виде – устройтесь поудобнее в вашем любимом кресле. Не претендуя на роль полного учебного руководства, эта часть предлагает скоростное введение в Perl, что устроит не всякого читателя. В разделе «Печатная документация» (ниже) поищите книги, которые лучше сочетаются с вашим стилем учебы.

Часть II «Анатомия Perl»

В этой части проводится глубокое и ничем не ограниченное обсуждение внутреннего устройства языка на всех уровнях абстракции – от типов данных, переменных и регулярных выражений до подпрограмм, модулей и объектов. Читатель получит хорошее представление о том, как работает язык, а также несколько советов по правильному проектированию программ. (А тех, кто никогда не использовал язык с поиском по шаблону, ждет особое удовольствие.)

Часть III «Perl как технология»

Многое можно делать с помощью одного только Perl, но в этой части вы изучите волшебство более высокого уровня. Узнаете о том, как заставить Perl пройти через все препятствия, которые поставит перед ним ваш компьютер, – от обработки Юникода, взаимодействия процессов и многопоточности до компилирования, вызова, отладки и профилирования, а также создания собственных внешних расширений на C или C++ или интерфейсов к имеющимся API. Perl будет счастлив побеседовать с любым интерфейсом на вашем компьютере, да, пожалуй, и любом другом компьютере в Интернете, если позволят погодные условия.

---

[1] Псевдохеши – виртуальные ассоциативные массивы. – *Прим. ред.*

[2] Воспользуйтесь командой *perldoc perldiag* или адресом *http://perldoc.perl.org/perldiag.html*. – *Прим. ред.*

Часть IV «Perl как культура»

Каждому ясно, что у культуры должен быть свой язык, но сообществу Perl всегда было ясно, что у языка должна быть культура. В этой части мы рассматриваем программирование на Perl как человеческую деятельность, являющуюся частью реального мира людей. Мы также даем много советов относительно того, как заниматься самосовершенствованием и как сделать, чтобы ваши программы приносили больше пользы людям.

Часть V «Справочный материал»

Здесь собраны главы, в которых читатель сможет найти что-либо в алфавитном порядке – от специальных переменных и функций до стандартных модулей и прагм. Глоссарий будет особенно полезен тем, кто не знаком с жаргоном вычислительной техники. Например, те, кто не знает, что такое «прагма», могут прямо сейчас посмотреть значение этого слова. (А тем, кто не знает значение слова «такое», мы не можем помочь ничем.)

Стандартный дистрибутив

Официальная политика Perl, как отмечается в странице *perlpolicy* справочного руководства[1], заключается в поддержке двух последних официальных версий. Поскольку на момент написания этих строк текущей была версия v5.14, это означает, что официально поддерживаются обе версии, v5.12 и v5.14. Когда будет выпущена версия v5.16, официальная поддержка версии v5.12 прекратится.

В настоящее время большинство производителей операционных систем включают Perl в качестве стандартной составляющей своей системы, хотя их цикл выпуска новых версий может не совпадать с циклом выпуска новых версий Perl. На момент написания данной книги Perl входит в стандартные дистрибутивы AIX, BeOS, BSDI, Debian, DG/UX, DYNIX/ptx, FreeBSD, IRIX, LynxOS, Mac OS X, OpenBSD, OS390, RedHat, SINIX, Slackware, Solaris, SuSE и Tru64. Некоторые компании поставляют Perl на отдельных CD с бесплатным программным обеспечением или через группы обслуживания клиентов. Сторонние производители, такие как ActiveState, предоставляют откомпилированные дистрибутивы для ряда операционных систем, в том числе производимых Microsoft.

Даже если производитель включил Perl в стандартный дистрибутив, в конечном итоге, возможно, понадобится откомпилировать и установить Perl самостоятельно. В результате вы будете знать, что ваша версия является самой свежей, и сможете сами выбрать, куда установить библиотеки и документацию. Также можно будет решить, следует ли скомпилировать Perl с поддержкой дополнительных расширений, таких как поддержка многопоточной модели выполнения, большие файлы или множество низкоуровневых опций отладки, доступ к которым осуществляется через ключ командной строки *-D*. (Отладчик уровня пользователя поддерживается всегда.)

Проще всего загрузить комплект исходного кода Perl, указав браузеру домашнюю страницу на *www.perl.org*, где на видном месте располагается информация о загружаемых файлах, а также доступны ссылки на компилированные двоичные модули для платформ, компиляторы C для которых затерялись.

---

[1] Команда *perldoc perlpolicy* или *http://perldoc.perl.org/perlpolicy.html*. – *Прим. ред.*

Можно также направиться прямо в архив CPAN, описанный в главе 19, по адресу *http://www.cpan.org*. Если работа с ним окажется слишком медленной (а это может случиться, поскольку он очень популярен), следует найти зеркальный сервер CPAN поблизости от себя. На странице *http://www.cpan.org/SITES.html* приводится список всех сайтов архива CPAN, откуда вы можете выбрать удобное для вас зеркало. Некоторые зеркала доступны по FTP, другие по HTTP (что может иметь значение для тех, кто выходит в Интернет из корпоративной сети, защищенной брандмауэром). Мультиплексор *http://www.cpan.org* попытается принять решение автоматически, однако при желании вы легко сможете изменить этот выбор.

Получив исходный код и распаковав его в каталог, следует прочесть файлы *README* и *INSTALL*, чтобы узнать, как выполнить сборку Perl. В каталоге может также иметься файл *INSTALL.platform*, где *platform* представляет платформу вашей операционной системы.

Если данная платформа является разновидностью UNIX, то команды, необходимые для получения, конфигурирования, сборки и установки Perl, могут быть примерно следующие. Во-первых, необходимо выбрать команду, с помощью которой будет получен исходный код. Загрузить пакет можно с помощью браузера или инструмента командной строки:

```
% wget http://www.cpan.org/src/5.0/maint.tar.gz
```

Теперь нужно распаковать, сконфигурировать, собрать и установить:

```
% tar zxf latest.tar.gz        # или сначала gunzip, а затем tar xf.
% cd perl-5.14.2               # или 5.* для других версий
% sh Configure -des            # принимает ответы по умолчанию.
% make test && make install    # обычно требует привилегий суперпользователя
```

Для вашей платформы могут иметься уже готовые пакеты, не требующие выполнения всех этих операций (а также включающие исправления и расширения для вашей платформы). Кроме того, многие платформы уже включают предустановленный Perl, так что описанные действия могут оказаться ненужными.

Если Perl уже установлен, но вам хочется установить другую версию, можно избежать лишней работы, воспользовавшись инструментом *perlbrew*. Он автоматизирует все описанные действия и выполняет установку в каталог, куда вы имеете право устанавливать файлы, если не обладаете привилегиями администратора. Этот инструмент доступен в CPAN под названием `App::perlbrew`, но вы можете установить его, выполнив следующие действия, как описывается в документации:

```
% curl -L http://xrl.us/perlbrewinstall | bash
```

После установки просто позвольте этому инструменту выполнить всю работу за вас:

```
% ~/perl5/perlbrew/bin/perlbrew install perl-5.14.2
```

Однако этим возможности инструмента *perlbrew* не ограничиваются, поэтому за дополнительной информацией обращайтесь к документации.

Существуют также расширенные версии стандартного дистрибутива Perl. Компания ActiveState предлагает ActivePerl (*http://www.activestate.com/activeperl/downloads*) – бесплатные версии для Windows, Mac OS X и Linux и платные – для Solaris, HP-UX и AIX.

Strawberry Perl (*http://strawberryperl.org/*) – версия для Windows, включающая различные инструменты, необходимые для компиляции и установки сторонних модулей Perl из CPAN.

Citrus Perl (*http://www.citrusperl.com/*) – дистрибутив для Windows, Mac OS X и Linux, включающий инструментарий wxPerl для создания графических интерфейсов. Он предназначен для тех, кто желает создавать на языке Perl программы с графическим интерфейсом. А в распространении этих приложений вам поможет другой инструмент, Cava Packager (*http://www.cava.co.uk/*), также входящий в состав этого дистрибутива.

Электронная документация

Обширная электронная документация по Perl входит в состав его стандартного дистрибутива. (О печатной документации говорится в следующем разделе.) Дополнительная документация появляется, как только устанавливается новый модуль из CPAN.

Упоминая в этой книге «страницы руководства Perl», мы имеем в виду комплект электронных страниц руководства по Perl, который находится на вашем компьютере. Под *страницей электронного руководства* (*manpage*) будем понимать просто файл с документацией, для чтения которого необязательно иметь UNIX-программу *man*. Страницы руководства Perl могут быть установлены даже как страницы HTML, особенно в системах, отличных от UNIX.

Электронные страницы руководства по Perl разделены на несколько секций, поэтому можно легко найти нужное, не продираясь через сотни страниц текста. Поскольку страница верхнего уровня называется просто *perl*, то в UNIX команда `man perl` должна привести именно на нее.[1] Эта страница, в свою очередь, обозначает страницы, посвященные конкретным темам. Например, `man perlre` выведет страницу руководства по регулярным выражениям Perl. Команда `perldoc` часто работает в тех системах, в которых не работает команда `man`. В вашем дистрибутиве могут также содержаться страницы руководства по Perl в формате HTML или родном для системы формате подсказки. Уточните этот вопрос у своего системного администратора – если, конечно, сами не являетесь им.

Навигация по стандартным страницам руководства

В незапамятные времена (если говорить о Perl, то имеется в виду 1987 год) страница руководства *perl* была кратким документом объемом около 24 печатных страниц. Например, раздел по регулярным выражениям занимал всего два абзаца. (Этого было достаточно при условии знакомства с *egrep*.) В некоторых отношениях с тех пор изменилось почти все. Если считать стандартную документацию, различные утилиты, сведения о переносах на различные платформы и множество стандартных модулей, то наберется несколько тысяч печатных страниц документации, разбросанных по многим отдельным страницам электронного руководства.

---

[1] Если при этом открывается нечто необозримое, то, вероятно, вы обращаетесь к древнему руководству версии 4. Проверьте, не указывает ли переменная среды `MANPATH` на места, где можно производить археологические раскопки. (Введите `perldoc perl`, чтобы узнать, как настроить `MANPATH` соответственно выдаче команды `perl -V:man.dir`.)

(И это без учета модулей из CPAN, которые могут быть у вас установлены, и в немалом количестве.)

Но в других отношениях не изменилось ничего: по-прежнему жива страница руководства *perl* и по-прежнему она является наилучшей отправной точкой, когда вы не знаете, откуда начать. Разница в том, что, попав туда, нельзя остановиться. Документация – это больше не кустарное производство, это торговый пассаж с сотнями магазинов. Войдя в дверь, необходимо найти плакат, который поможет определить, где находится лавка или универсальный магазин, продающий то, зачем вы пришли. Конечно, освоившись в торговом центре, как правило, заранее знаешь, куда направиться.

Вот некоторые вывески:

Таблица 1. Некоторые страницы руководства по Perl

Страница руководства	Что освещает
perl	Доступные страницы руководства по Perl
perldata	Типы данных
perlsyn	Синтаксис
perlop	Операторы и их приоритеты
perlre	Регулярные выражения
perlvar	Предопределенные переменные
perlsub	Подпрограммы
perlfunc	Встроенные функции
perlmod	Как использовать модули Perl
perlref	Ссылки
perlobj	Объекты
perlipc	Взаимодействие между процессами
perlrun	Как выполнять команды Perl, а также ключи
perldebug	Отладка
perldiag	Диагностические сообщения

Это лишь небольшой отрывок, но в нем содержится самое необходимое. Как видно, если нужны сведения о том или ином операторе, вам потребуется *perlop*. А если нужно что-то выяснить о предопределенных переменных, следует искать в *perlvar*. Если получено непонятное диагностическое сообщение, обращайтесь к *perldiag*. И так далее.

В стандартное руководство по Perl входит список часто задаваемых вопросов (FAQ). Он разбит на девять разных страниц, перечисленных в табл. 2:

Таблица 2. Страницы из списка часто задаваемых вопросов

Страница руководства	Что освещает
perlfaq1	Общие вопросы о Perl
perlfaq2	Получение Perl и сведения о нем
perlfaq3	Инструменты программирования

Таблица 2 (продолжение)

Страница руководства	Что освещает
perlfaq4	Обработка данных
perlfaq5	Файлы и форматы
perlfaq6	Регулярные выражения
perlfaq7	Общие особенности языка Perl
perlfaq8	Взаимодействие с системой
perlfaq9	Сетевое взаимодействие

Некоторые страницы руководства (табл. 3) содержат замечания по специфике платформ:

Таблица 3. Страницы с замечаниями по специфике платформ

Страница руководства	Что освещает
perlamiga	Версия для Amiga
perlcygwin	Версия для Cygwin
perldos	Версия для MS-DOS
perlhpux	Версия для HP-UX
perlmachten	Версия для Power MachTen
perlos2	Версия для OS/2
perlos390	Версия для OS/390
perlvms	Версия для DEC VMS
perlwin32	Версия для MS-Windows

Относительно версий для разных платформ см. также главу 22 и каталог ports в архиве CPAN (*http://www.cpan.org/ports/index.html*), описанном выше.

Страницы руководства, не принадлежащие Perl

Когда мы ссылаемся на документацию, не принадлежащую Perl, как в случае *getitimer*(2), ссылка указывает на страницу *getitimer* из раздела 2 руководства Unix Programmer's Manual.[1] Страницы руководства для системных вызовов, вроде *getitimer*, могут отсутствовать в системах, отличных от UNIX, но ведь системные вызовы UNIX в таких операционных системах все равно использовать не получится. Для тех, кому действительно требуется документация по команде, системному вызову или библиотечной функции UNIX, скажем, что многие организации поместили свои страницы руководства в Интернете. Выполнив поиск по строке `crypt(3)` в Google, можно найти множество копий данной конкретной страницы.

[1] В разделе 2 должны содержаться только сведения о прямых вызовах операционной системы. (Они часто называются *системными обращениями* («system calls»), но мы неуклонно называем их в этой книге *системными вызовами* (syscalls), чтобы не спутать с функцией `system`, не имеющей отношения к системным вызовам. Однако между системами есть некоторые различия в том, какие вызовы реализованы как системные, а какие – как обращения к библиотекам C, поэтому есть вероятность, что `getitimer(2)` обнаружится в разделе 3.

Хотя страницы руководства по Perl верхнего уровня обычно устанавливаются в разделе 1 стандартных каталогов *man*, мы не будем добавлять (1) к названиям таких страниц руководства в нашей книге. Их легко узнать, поскольку они имеют вид «perl*бубубу*».

Печатная документация

Тем, кто хочет больше узнать о Perl, рекомендуем некоторые издания:

- «Perl 5 Pocket Reference», by Johan Vromans; O'Reilly Media (5th Edition, July 2011). Эта маленькая брошюра служит удобным справочником по Perl.
- «Perl Cookbook», by Tom Christiansen and Nathan Torkington, O'Reilly Media (2nd Edition, August 2003).[1] Эта книга дополняет ту, которая сейчас у вас в руках. Она содержит готовые рецепты программирования на Perl.
- «Learning Perl», by Randal Schwartz, brian d foy, and Tom Phoenix; O'Reilly Media (6th Edition, June 2011).[2] Эта книга учит программистов тем 30% основ Perl, которые пригождаются в 70% случаев. Она ориентирована на тех, кто пишет автономные программы, содержащие порядка пары сотен строк.
- «Intermediate Perl», by Randal Schwartz, brian d foy, and Tom Phoenix; O'Reilly Media (August 2012).[3] Эта книга является продолжением книги «Learning Perl» («Изучаем Perl») и знакомит читателей со ссылками, структурами данных, пакетами, объектами и модулями.
- «Mastering Perl», by brian d foy; O'Reilly Media (July 2007). Это последняя книга трилогии, включающей также книги «Learning Perl» («Изучаем Perl») и «Intermediate Perl» («Изучаем Perl глубже»). Вместо основ языка она рассматривает вопросы применения Perl для решения повседневных задач.
- «Modern Perl», by chromatic; Oynx Neon (October 2010). Эта книга содержит обзор современных тем и приемов программирования на Perl для тех, кто уже занимается программированием, но не уделяет внимания последним событиям в мире Perl.
- «Mastering Regular Expressions», by Jeffrey Friedl; O'Reilly Media (3rd Edition, August 2006).[4] Хотя в книге не освещены последние нововведения в регулярных выражениях Perl, она является ценным справочником для всех, кто хочет знать, как работают регулярные выражения.
- «Object Oriented Perl», by Damian Conway, Manning, 1999. Для начинающих и опытных разработчиков объектно-ориентированных программ. Эта книга излагает обычные и тайные приемы создания мощных объектных систем на Perl.

---

1 Т. Кристиансен, Н. Торкингтон «Perl: библиотека программиста». – Пер. с англ. – СПб.: Питер, 2001.

2 Р. Шварц, брайан д фой и Т. Феникс «Изучаем Perl», 5-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2009.

3 Р. Шварц, брайан д фой и Т. Феникс «Perl: изучаем глубже», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2014.

4 Д. Фридл «Регулярные выражения», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008.

- «Mastering Algorithms with Perl», by Jon Orwant, Jarkko Hietaniemi, and John Macdonald, O'Reilly, 1999. Все полезные приемы из курса вычислительных алгоритмов, но без тягостных доказательств. Книга освещает фундаментальные и полезные алгоритмы, относящиеся к графам, текстам, множествам и ряду других областей.

Существует много других книг и публикаций по Perl, и по старческой дряхлости мы наверняка позабыли упомянуть некоторые хорошие. (Из милосердия мы не стали отмечать некоторые плохие издания.)

Помимо перечисленных публикаций, относящихся к Perl, мы рекомендуем следующие книги. Они не посвящены непосредственно Perl, но их бывает удобно иметь под рукой в качестве справочника, источника советов или вдохновения.

- «The Art of Computer Programming», by Donald Knuth, Volumes 1–4A: «Fundamental Algorithms», «Seminumerical Algorithms», «Sorting and Searching», and «Combinatorial Algorithms»; Addison-Wesley (2011).[1]
- «Introduction to Algorithms», by Thomas Cormen, Charles Leiserson, and Ronald Rivest; MIT Press and McGraw-Hill (1990).
- «Algorithms in C», by Robert Sedgewick, Addison-Wesley (1990).
- «The Elements of Programming Style», by Brian Kernighan and P.J. Plauger; Prentice-Hall (1988).
- «The Unix Programming Environment», by Brian Kernighan and Rob Pike; Prentice-Hall (1984)[2] (3rd Edition, August 2006).
- «POSIX Programmer's Guide», by Donald Lewine, O'Reilly Media (1991).
- «Advanced Programming in the UNIX Environment», by W. Richard Stevens and Stephen A. Rago, Addison-Wesley (3rd Edition, May 2013).[3]
- «TCP/IP Illustrated», vols. 1–3, by W. Richard Stevens, Addison-Wesley (1992–1996).
- «The Lord of the Rings», by J. R. R. Tolkien, Houghton Mifflin (U.S.) and Harper Collins (U.K.) (последнее издание: 2005).[4]

Дополнительные источники

Интернет – чудесное изобретение, и все мы продолжаем открывать возможности его наиболее полного использования. (Конечно, некоторые предпочитают «открывать» Интернет так, как Толкин открывал Средиземье.)

[1] Д. Кнут «Искусство программирования», тома 1–4А: «Основные алгоритмы», «Получисленные алгоритмы», «Сортировка и поиск», «Комбинаторные алгоритмы, часть 1». – Пер. с англ. – Вильямс, 2013.

[2] Брайан Керниган и Роб Пайк «UNIX. Программное окружение». – Пер. с англ. – СПб.: Символ-Плюс, 2003.

[3] У. Ричард Стивенс и Стивен Раго «UNIX. Профессиональное программирование», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2014.

[4] Дж. Р.Р. Толкин «Властелин Колец», АСТ, 2009.

Perl в Сети

Посетите домашнюю страницу Perl *http://www.perl.org/*. Она предлагает новости мира Perl, исходные коды и скомпилированные версии для разных платформ, тематические статьи, документацию, расписания конференций и многое другое.

Посетите также веб-страницу Perl Mongers *http://www.pm.org*, чтобы взглянуть на Perl с точки зрения широких масс, так сказать, на «корни», которые бурно разрастаются во всех частях света, за исключением Южного полюса, где их приходится держать в закрытом помещении. Местные группы PM проводят регулярные небольшие встречи, на которых можно обменяться профессиональным опытом с другими хакерами Perl, живущими в той же части света.

Сообщения об ошибках

В том маловероятном случае, если ошибка обнаружена в самом Perl, а не в вашей программе, нужно постараться воспроизвести ее в минимальном по объему контрольном примере и сообщить о ней с помощью программы *perlbug*, поставляемой с Perl. Дополнительные сведения можно найти на *http://bugs.perl.org*.

В действительности, команда *perlbug* является лишь интерфейсом к инструменту учета дефектов под названием RT.[1] Кроме того, отчет об ошибке можно отправить на адрес *perlbug@perl.org* без привлечения дополнительных инструментов, однако *perlbug* вместе с отчетом отправляет дополнительную информацию о вашей версии Perl, такую как номер версии и флаги компиляции, которые могут помочь разработчикам определить источник вашей проблемы.

Можно также заглянуть в список существующих дефектов. Возможно, кто-то уже столкнулся с вашей проблемой. Начните со страницы *https://rt.perl.org/* и перейдите по ссылке *perl5*.

Если речь идет о стороннем модуле из архива CPAN, следует воспользоваться другой версией инструмента RT по адресу: *https://rt.cpan.org/*. Однако следует учитывать, что не все модули в архиве CPAN пользуются бесплатной услугой RT, поэтому всегда проверяйте документацию, где могут приводиться дополнительные инструкции, касающиеся отправки отчетов об ошибках.

Соглашения, принятые в этой книге

Некоторые из принятых нами соглашений более подробно обсуждаются в соответствующих разделах. Соглашения по коду обсуждаются в разделе «Стиль программирования» главы 21. Можно сказать, что наши лексические соглашения представлены в глоссарии (наш словарь).

В книге приняты следующие типографские соглашения:

ЗАГЛАВНЫЕ СИМВОЛЫ

Используются для обозначения формальных названий символов Юникода и логических операторов.

---

[1] Проект Best Practical, в рамках которого осуществляется разработка инструмента Request Tracker, или RT, бесплатно предоставляет свою платформу основным проектам Perl, включая сам *perl* и все пакеты в архиве CPAN.

Курсив

Применяется для URL-адресов, страниц руководства, имен файлов и программ. Новые термины тоже выделяются курсивом при первом появлении в тексте. Для многих из этих терминов можно найти альтернативные определения в глоссарии, если недостаточно тех, которые имеются в тексте. Курсив также используется для выделения команд и ключей командной строки. Это, к примеру, позволяет отличить ключ *-w*, включающий вывод предупреждений, от параметра -w (проверки наличия файла).

Моноширинный

Используется в примерах и в обычном тексте для выделения программного кода. Данные обозначаются моноширинным шрифтом и заключаются в кавычки (""), не являющиеся частью значения.

Моноширинный курсив

Служит для обозначения общих элементов кода, вместо которых необходимо подставить конкретные значения. Он также используется в некоторых примерах, чтобы выделить вывод, производимый программой.

Моноширинный полужирный

Этот шрифт используется в примерах для обозначения текста, который вводится в командной строке.

Моноширинный полужирный курсив

Используется для выделения вывода программы, когда требуется отличать его от текста, вводимого в командной строке.

Мы приводим множество примеров, и в большинстве случаев это фрагменты более крупных программ. Некоторые примеры являются завершенными программами, что можно увидеть по начальной строке #!. Почти все наши более длинные программы начинаются со строки:

```
#!/usr/bin/perl
```

В других примерах приводится текст, который следует ввести в командной строке. Мы используем % для обозначения приглашения оболочки:

```
% perl -e 'print "Hello, world.\n"'
Hello, world.
```

Этот стиль – типичный образец стандартной командной строки UNIX, в которой одиночные кавычки представляют «наиболее закавыченный» формат. В других системах соглашения по использованию кавычек и шаблонных символов могут быть иными. Например, многие интерпретаторы команд в MS-DOS и VMS требуют двойные, а не одиночные кавычки при определении аргументов, содержащих пробелы и шаблонные символы.

Благодарности

Здесь мы публично приносим благодарность нашим советникам, консультантам и рецензентам, что должно компенсировать все грубости, сказанные им в частном порядке. Это: Эбигайл (Abigail), Мэтью Барнетт (Matthew Barnett), Пирс Коули (Piers Cawley), chromatic (хроматик), Дамиан Конвей (Damian Conway), Дейв Кросс

(Dave Cross), Хоакин Ферреро (Joaquin Ferrero), Джеремиа Фостер (Jeremiah Foster), Джефф Хеймер (Jeff Haemer), Юрий Маленький (Yuriy Malenkiy), Нуно Мендес (Nuno Mendes), Стеффен Мюллер (Steffen Müller), Энрике Нелл (Enrique Nell), Дэвид Никол (David Nicol), Флориан Рагвитц (Florian Ragwitz), Эллисон Рэндал (Allison Randal), Крис Роедер (Chris Roeder), Кит Томпсон (Keith Thompson), Леон Тиммерманс (Leon Timmermans), Натан Торкингтон (Nathan Torkington), Йохан Вроманс (Johan Vromans) и Карл Уильямсон (Karl Williamson).

Особую благодарность хотим выразить всем сотрудникам издательства O'Reilly Media за их героические усилия в преодолении бесчисленных, неожиданных и необычных проблем, связанных с опубликованием этой книги в современном постмодернистском издательском мире. В первую очередь мы хотели бы сказать спасибо нашему выпускающему редактору Холли Бауэр (Holly Bauer) за ее бесконечное терпение в отношении тысяч маленьких уточнений и изменений, сделанных уже после сдачи рукописи. Мы благодарны начальнику производственного отдела Дэну Фоксмиту (Dan Fauxsmith) за его упорство в поиске редких шрифтов, необходимых для многочисленных примеров использования Юникода, и за поддержание плавности хода производственного конвейера. Мы признательны заведующему производством Адаму Уитверу (Adam Witwer) за то, что он, засучив рукава, самоотверженно боролся с программным комплексом Antenna House, применявшимся для создания готовой к тиражированию копии книги. Наконец, благодарим издателя Лори Петрицки (Laurie Petrycki): она не просто поддержала всех причастных к созданию книги, о которой мечтали авторы, но и воодушевила самих авторов на создание книг, прочтение которых приносит людям массу удовольствия.

Safari® Books Online



Safari Books Online – это виртуальная библиотека, которая позволяет легко и быстро находить ответы на вопросы среди более чем 7500 технических и справочных изданий и видеороликов.

Подписавшись на услугу, вы сможете загружать любые страницы из книг и просматривать любые видеоролики из нашей библиотеки. Читать книги на своих мобильных устройствах и сотовых телефонах. Получать доступ к новинкам еще до того, как они выйдут из печати. Читать рукописи, находящиеся в работе, и посылать свои отзывы авторам. Копировать и вставлять отрывки программного кода, работать с закладками, загружать отдельные главы, помечать ключевые разделы, оставлять примечания, печатать страницы и пользоваться массой других преимуществ, позволяющих экономить ваше время.

Благодаря усилиям O'Reilly Media данная книга также доступна через службу Safari Books Online. Чтобы получить полный доступ к электронной версии этой книги, а также к другим книгам схожей тематики, изданным O'Reilly и другими издательствами, подпишитесь бесплатно по адресу *http://my.safaribooksonline.com*.

Хотим услышать ваши отзывы

Мы протестировали и выверили все сведения, представленные в этой книге так тщательно, как могли, но читатели могут обнаружить, что некоторые функции

претерпели изменения (или даже что мы допустили ошибки!). Просим сообщать обо всех найденных ошибках, а также пожеланиях для последующих изданий по адресу:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
1-800-998-9938 (в США и Канаде)
1-707-829-0515 (международный/местный)
1-707-829-0104 (факс)

Книга имеет свой веб-сайт, где можно найти список обнаруженных ошибок и другую информацию:

http://shop.oreilly.com/product/9780596004927.do

Здесь же вы найдете все примеры программного кода, доступные для загрузки, что избавит вас от необходимости набирать их вручную, как это делали мы.

Свои комментарии и вопросы технического характера, касающиеся этой книги, направляйте по адресу:

bookquestions@oreilly.com.

Дополнительную информацию о книгах, конференциях, центрах ресурсов и сети издательства O'Reilly вы найдете на сайте:

http://www.oreilly.com

I

Общий обзор

1

Обзор Perl

Введение

Мы считаем Perl языком, который легко освоить и применять, и надеемся убедить в этом и вас. Одна из особенностей Perl, делающих его легким языком, заключена в возможности без обиняков сказать то, что необходимо сказать. Во многих языках программирования приходится объявлять типы, переменные и подпрограммы, которые предполагается использовать, прежде чем написать хотя бы одну строчку выполняемого кода. Подобные объявления – разумный подход, если речь идет о сложных задачах, требующих сложных структур данных. Но для многих более простых, повседневных задач хотелось бы иметь язык программирования, на котором можно просто сказать:

```
print "Howdy, world!\n";
```

и программа это сделает.

Таким языком и является Perl. На практике этот пример представляет собой законченную программу[1], и если передать ее интерпретатору Perl, тот выведет на экран `"Howdy, world!"`. (`\n` в этом примере обеспечивает перевод строки в конце вывода.)

Вот и все. *После* того как вы сказали, что хотели, тоже не требуется произносить много слов. В отличие от многих языков программирования, Perl считает, что «вывалиться» из программы, когда закончился исходный текст, – нормальный способ ее завершения. Конечно, при желании вы можете явно вызвать функцию `exit`, как *можете* и объявить некоторые переменные, либо даже сделать *обязательным* объявление всех переменных. Но право выбора принадлежит вам. В Perl можно поступать «так, как должно» согласно вашему личному пониманию этих слов.

Есть много других причин, по которым Perl легко использовать, но перечислять сейчас их все бессмысленно, поскольку этому и посвящена наша книга. Говорят,

---

[1] Или «сценарий» (script), «приложение» (application), «исполняемый объект» (executable), «штуковина» (doohickey). Кому что больше нравится.

что дьявол проявляет себя в частностях, но Perl старается облегчить вашу жизнь и в трудных ситуациях. На любом уровне Perl заботится о том, чтобы доставить вас туда, куда нужно, с минимумом суеты и максимумом комфорта. Вот почему многие программисты на Perl ходят с глупыми ухмылками на физиономиях.

Эта глава представляет собой обзор, поэтому мы не пытаемся представить Perl с рациональной стороны. Равно как не претендуем на полноту изложения или логичность. Для этого предназначены следующие главы. Вулканцам, андроидам и другим лицам аналогичного склада ума следует пропустить этот обзор и перейти сразу к главе 2, где плотность информации выше. С другой стороны, тем, кому нужен учебник, где все постепенно раскладывается по полочкам, возможно, следует обратиться к книге «Learning Perl»[1]. Но и данную книгу пока не выбрасывайте.

Эта глава представляет Perl *другому* полушарию вашего мозга, тому, которое называют ассоциативным, артистическим, управляющим эмоциями или просто впитывающим. Для этого Perl рассматривается с разных точек зрения, чтобы дать читателю о нем такое же ясное представление, какое слепцы могли получить о слоне (ощупывая его). Возможно, мы продвинемся дальше мудрецов: мы все-таки имеем дело с верблюдом (см. обложку). Надеемся, что хотя бы одно из этих представлений о Perl поможет вам меньше горбатиться.

Естественные и искусственные языки

Языки были изобретены людьми и для блага людей. В анналах компьютерных наук этот факт случайно оказался позабытым.[2] Поскольку (допуская вольность речи) можно сказать, что создателем Perl является лингвист, этот язык, по задумке, должен работать так же гладко, как естественные языки. Само собой, это охватывает множество аспектов, поскольку естественные языки хорошо работают одновременно на нескольких уровнях. Можно было бы перечислить многие из этих лингвистических принципов, но самым важным принципом проектирования языка является тот, что простые вещи должны делаться просто, а сложные вещи должны быть реализуемы. (На самом деле, здесь два принципа.) Они могут показаться вам очевидными, однако многие языки программирования не удовлетворяют то одному, то другому условию.

Естественные языки удовлетворяют обоим условиям, поскольку люди непрерывно пытаются выражать как простые, так и сложные мысли, и поэтому языки развиваются так, чтобы справляться с обеими задачами. Perl проектировался, прежде всего, как язык развивающийся, и он действительно эволюционировал со времени своего создания. Вклад в эволюцию Perl вносили многие люди на протяжении ряда лет. Мы часто шутим, что верблюд – это лошадь, спроектированная комитетом, но если вдуматься, верблюд достаточно хорошо приспособлен для жизни в пустыне. Эволюция привела верблюда к относительной самодостаточности. (С другой стороны, в результате эволюции верблюд не стал лучше пахнуть. Как и Perl.) Вот одна из многих странных причин, почему мы сделали верблюда талисманом Perl, но к лингвистике это имеет отдаленное отношение.

---

1 Рэндал Шварц, Том Феникс, брайан д фой «Изучаем Perl», 5-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2009.

2 Точнее, об этом факте вспоминают от случая к случаю.

Далее. Когда кто-то произносит слово «лингвистика», многие начинают думать о словах либо о предложениях. Но слова и предложения – это лишь пара удобных способов членения речи. Те и другие можно разбить на более мелкие смысловые единицы или объединить в более крупные. А смысл каждой единицы существенно зависит от синтаксического, семантического и практического контекста, в котором находится эта единица. В естественном языке есть слова разных видов: существительные, глаголы и прочее. Если произнести слово «dog» (собака) изолированно, в отрыве от контекста, представляется существительное, хотя можно использовать это слово и в других смыслах. То есть существительное может выполнять функции глагола, прилагательного или наречия в зависимости от контекста. «If you dog a dog during the dog days of summer, you'll be a dog tired dogcatcher.»[1] Perl также по-разному интерпретирует слова в зависимости от контекста. Далее мы увидим, как он это делает. Помните просто, что Perl пытается понять, о чем идет речь, как это делает любой хороший слушатель. Perl со своей стороны усиленно старается соблюсти условия сделки. Скажите, чего вы хотите, а Perl, как правило, «сообразит». (Если только вы не начнете нести бессмыслицу – анализатор Perl понимает Perl значительно лучше, чем английский или суахили.)

Но вернемся к существительным. Существительное может быть названием конкретного объекта или родового класса объектов – и само по себе оно не содержит информации о том, который из вариантов используется. Это различие проводится в большинстве языков программирования, когда конкретная величина называется значением, а общая – переменной. Значение где-то просто существует, а вот переменная в течение срока своей жизни связывается с одним или более значений. Поэтому механизм интерпретации обязан отслеживать связь переменной с конкретными значениями. Интерпретатор может находиться в мозгу человека или в компьютере.

Синтаксис переменных

Переменная – это просто удобный способ хранить что-либо, именованный ящик, в котором программист может найти нечто требующееся ему, когда оно позднее понадобится. Как и в реальной жизни, есть ячейки хранения разного типа: одни являются, по существу, тайными, а другие открыты для всех. Некоторые хранилища временные, а другие – более постоянные. Специалисты в области информатики любят обсуждать «области видимости» переменных, подразумевая под этим выражением именно то, что в нем сказано. В Perl имеются достаточно удобные средства решения проблем области видимости, с которыми читатель в будет иметь удовольствие познакомиться в должное время. Оно еще не наступило. (Лю-

---

[1] Пример многозначности слова «dog» и зависимости от контекста. Здесь оно используется в роли глагола; в роли существительного; в составе идиомы, которая вообще переводится без применения слова «собака» (dog days, дождливые дни или дни повышенной влажности); в составном слове dogcatcher, которое может означать сотрудника ведомства по надзору за дикими животными или выражать презрительное отношение к бестолковому политику; а также для описания сильной усталости (устал как собака). – *Прим. перев.*

Возможно, читатель устал как собака от всей этой лингвистической трескотни. Но мы хотим, чтобы было понятно, чем Perl, чтоб его собаки подрали, отличается от обычных компьютерных языков!

бопытные могут поискать прилагательные local, my, our и state в главе 27 либо заглянуть в раздел «Объявления, ограниченные областью видимости» главы 4.)

В данный момент полезнее провести классификацию переменных по типам данных, которые они способны хранить. Как и в английском языке, в Perl основное различие проводится между данными в единственном и во множественном числе. Строки и числа являются данными в единственном числе, а списки строк или чисел – во множественном. (А когда мы доберемся до объектно-ориентированного программирования, вы обнаружите, что типичный объект выглядит единичным снаружи, но множественным изнутри – как группа студентов.) Переменную в единственном числе мы называем *скаляром*, а во множественном – *массивом*. Поскольку строка может храниться в скалярной переменной, можно написать такую, несколько более длинную (и снабженную комментариями) версию нашего первого примера:

```
my $phrase = "Howdy, world!\n"; # Создать новую переменную.
print $phrase;                  # Вывести переменную.
```

Прилагательное my сообщает интерпретатору Perl, что $phrase – совершенно новая переменная, поэтому ему не нужно пытаться искать переменную с этим именем. Обратите внимание, что нам не пришлось предварительно определять, какой тип имеет переменная $phrase. Символ $ указывает Perl на то, что phrase представляет собой скалярную переменную, т.е. содержит единственное значение. Напротив, имя переменной-массива должно начинаться символом @. (Возможно, вам будет проще запомнить, что $ – это стилизованная буква «s», от «scalar», тогда как @ – стилизованная «a», от «array».)[1]

В Perl имеются некоторые другие типы переменных с такими непривлекательными именами, как «хеш» (hash), «дескриптор» (handle) и «запись таблицы имен» (typeglob). Как и в случае скаляров и массивов, переменным этих типов предшествуют забавные символы, которые более широко известны как разыменовывающие префиксы. В табл. 1.1 приводится полный список разыменовывающих префиксов, которые могут вам встретиться:

Таблица 1.1. Типы переменных и области их использования

Тип	Символ	Пример	Что обозначает
Скаляр	$	$cents	Отдельное значение (число или строку)
Массив	@	@large	Список значений с числовым ключом
Хеш	%	%interest	Группа значений со строковым ключом
Подпрограмма	&	&how	Фрагмент кода Perl, который может быть вызван
Typeglob	*	*struck	Все, что имеет имя struck

Некоторые пуристы от языков программирования считают эти разыменовывающие префиксы причиной отвращения к Perl. Это поверхностный взгляд. У этих символов много достоинств, не последним из которых является возможность подстановки, или интерполяции (interpolation), этих символов в строки без всякого дополнительного синтаксиса. Кроме того, сценарии Perl легко читать (тем, кто

[1] Это упрощенное изложение настоящей истории разыменовывающих префиксов, которая будет рассказана в главе 2.

потрудился изучить Perl!), поскольку существительные отличаются от глаголов. В язык могут добавляться новые глаголы без ущерба для старых сценариев. (Мы уже говорили, что в Perl на этапе проектирования была заложена возможность развития.) И аналогия с существительными приведена не зря: есть масса примеров в английском и других языках, когда требуются грамматические маркеры существительных. Вот так мы думаем! (Как нам кажется.)

Единственное число

В приведенном выше примере мы видели, что скаляру можно присваивать новое значение с помощью оператора = подобно тому, как это делается во многих других языках программирования. Скалярным переменным можно присваивать скалярные значения любого вида: целые числа, числа с плавающей запятой, строки и даже такие эзотерические вещи, как ссылки на другие переменные или объекты. Есть много способов создания этих значений, которые должны присваиваться.

Как и в командной оболочке UNIX[1], можно использовать различные механизмы расстановки кавычек для создания значений разного вида. Двойные кавычки (double quotes) выполняют *интерполяцию переменных*[2] и *интерполяцию обратного слэша* (например, превращение \n в символ перевода строки), тогда как одиночные кавычки подавляют интерполяцию. А обратные кавычки (наклоненные влево) вызывают выполнение внешней программы и возврат ее выдачи, которая перехватывается как одна строка, содержащая все строки выдачи.

```
my $answer = 42;                # целое число
my $pi = 3.14159265;            # вещественное число
my $avocados = 6.02e23;         # экспоненциальная запись
my $pet = "Camel";              # строка
my $sign = "I love my $pet";    # строка с интерполяцией
my $cost = 'It costs $100';     # строка без интерполяции
my $thence = $whence;           # значение другой переменной
my $salsa = $moles * $avocados; # гастрохимическое выражение
my $exit = system("vi $file");  # числовой код результата выполнения команды
my $cwd = `pwd`;                # строка выдачи команды
```

И хотя мы еще не рассказали о необычных значениях, следует указать, что скаляры могут содержать ссылки на другие структуры данных, в том числе подпрограммы и объекты.

```
my $ary = \@myarray;            # ссылка на именованный массив
my $hsh = \%myhash;             # ссылка на именованный хеш
my $sub = \&mysub;              # ссылка на именованную подпрограмму

my $ary = [1,2,3,4,5];          # ссылка на анонимный массив
```

---

[1] Здесь и всюду, говоря «UNIX», мы подразумеваем любую операционную систему, похожую на UNIX, в том числе BSD, Mac OS X, Linux, Solaris, AIX и, конечно, UNIX.

[2] Иногда создатели сценариев командной оболочки называют это «подстановкой», но мы предпочитаем сохранить это слово для использования в Perl в других целях. Поэтому, пожалуйста, называйте это интерполяцией. Мы используем этот термин в текстологическом смысле («этот отрывок является гностической интерполяцией»), а не в математическом («данная точка на графике получена интерполяцией между двумя другими точками»).

```
my $hsh = {Na => 19, Cl => 35}; # ссылка на анонимный хеш
my $sub = sub { print $state }; # ссылка на анонимную подпрограмму
my $fido = Camel->new("Amelia");# ссылка на объект
```

Когда создается новая скалярная переменная, но до того как ей присвоено значение, она автоматически инициализируется значением undef, которое, как и следовало ожидать, означает «не определено» (undefined). В зависимости от контекста, неопределенное значение может интерпретироваться как пустое значение: "" или 0. В более общем случае, в зависимости от контекста использования, переменные автоматически интерпретируются как строки, числа или значения «true» (истина) или «false» (ложь), обычно называемые булевыми, или логическими значениями. Вспомните, как важен контекст в естественном языке. В Perl разные операторы предполагают получение в качестве параметров различного вида одиночных значений, и мы будем говорить, что данные операторы предоставляют для этих параметров скалярный контекст. Иногда мы будем более конкретно говорить, что оператор предоставляет числовой, строковый или логический контекст для этих параметров. (Ниже мы расскажем о списочном контексте, который противопоставляется скалярному.) Perl автоматически преобразует данные к виду, требуемому текущим контекстом, руководствуясь здравым смыслом. Предположим, например, что вы сказали следующее:

```
my $camels = '123';
print $camels + 1, "\n";
```

Первоначальным значением переменной $camels является строка, но она преобразуется в число, чтобы можно было прибавить к ней единицу, а затем обратно в строку, чтобы можно было вывести ее как 124. Символ перевода строки, "\n", тоже находится в строковом контексте, но поскольку он изначально является строкой, преобразование не требуется. Обратите, однако, внимание на двойные кавычки: использование одиночных кавычек, т.е. '\n', привело бы к появлению строки из двух символов, а именно обратного слэша и «n», которая никак не может представлять перевод строки.

Итак, в некотором смысле, двойные и одиночные кавычки предоставляют еще один способ уточнения контекста. Интерпретация содержимого строки в кавычках зависит от того, какого вида кавычки используются. (Далее мы рассмотрим некоторые другие операторы, синтаксически работающие как кавычки, но использующие строку некоторым особым способом, например, для поиска по шаблону или подстановки. Все они тоже действуют как строки в двойных кавычках. Контекст *двойных кавычек* является в Perl «интерполирующим» и предоставляется многими операторами, которые совершенно не похожи на двойные кавычки.)

Аналогично ссылка ведет себя как ссылка в «разыменовывающем» контексте и как обычная скалярная величина в других случаях. Например, можно сказать так:

```
my $fido = Camel->new("Amelia");
if (not $fido) { die "dead camel"; }
$fido->saddle();
```

Здесь создается ссылка на объект Camel, которая помещается в переменную $fido. В следующей строке мы проверяем значение $fido как логической величины на равенство «true» и возбуждаем исключительную ситуацию (т.е. жалуемся, что величина не истинна); в данном случае это означает, что конструктор Camel->new не смог создать требуемый объект Camel. Однако в последней строке мы обращаемся

с `$fido` как со ссылкой, вызывая метод `saddle()` объекта, содержащегося в `$fido` и представляющего собой объект `Camel`, поэтому Perl ищет метод `saddle()` для объектов Camel. Подробнее об этом сказано ниже. Сейчас просто запомните, что в Perl важен контекст, поскольку благодаря ему Perl узнает о ваших потребностях, не ожидая явных указаний, необходимых во многих других языках программирования.

Множественное число

В некоторых типах переменных содержатся множественные значения, связанные вместе логически. В Perl есть два типа многозначных переменных: массивы и хеши. Во многих случаях они ведут себя как скаляры: новые можно создать, например, с помощью объявления `my`, и они автоматически инициализируются пустым значением. Но в отличие от скаляров, при присвоении значений многозначным переменным в правой части оператора присваивания контекст является *списочным*, а не скалярным.

Массивы и хеши также различаются между собой. Массив следует использовать, когда нужно найти значение по его порядковому номеру. Хеш применяется, чтобы найти что-либо по названию. Эти два понятия дополняют друг друга. Часто можно наблюдать использование массива для перевода номера месяца в название и соответствующего хеша для перевода названия месяца обратно в номер. (Однако хеши могут содержать не только числа. К примеру, можно создать хеш, который будет переводить названия месяцев в названия камней, соответствующих месяцам.)

Массивы. *Массив* – это упорядоченный список скаляров, к которым обращаются по их местоположению в списке[1]. Список может содержать числа, строки или смесь того и другого. (Он может также содержать ссылки на вложенные массивы (субмассивы) и вложенные хеши (субхеши).) Чтобы присвоить массиву значение в виде списка, нужно просто объединить значения с помощью круглых скобок:

```
my @home = ("кушетка", "стул", "стол", "печка");
```

И наоборот, используя `@home` в списочном контексте, например, в правой части присваивания списку, мы получим на выходе тот самый список, который ввели. Например, присвоить значения, взятые из массива, четырем скалярным переменным можно так:

```
my ($potato, $lift, $tennis, $pipe) = @home;
```

Это называется списочным присваиванием. Логически оно происходит параллельно, поэтому можно обменять значениями две переменные с помощью такой фразы:

```
($alpha,$omega) = ($omega,$alpha);
```

Как и в языке C, нумерация элементов массивов начинается с нуля, поэтому, работая с элементами массива с первого по четвертый, следует обращаться к ним по индексам от 0 до 3.[2] Индексы массивов заключаются в квадратные скобки,

---

[1] Или по ключу (порядковому номеру, нижнему индексу), определяющему местоположение. Выберите вариант по душе.

[2] Если это кажется вам непривычным, считайте индекс смещением, т.е. числом предшествующих элементов массива. Очевидно, перед первым элементом нет других элементов, поэтому его смещение равно 0. Так уж думают компьютеры. (Как нам кажется.)

[вот такие], поэтому при выборе отдельного элемента массива нужно ссылаться на него так – $home[*n*], где *n* является значением индекса (на единицу меньшим номера элемента), который нам нужен. Посмотрите на следующий пример. Поскольку мы имеем дело со скалярным элементом, ему обязательно должен предшествовать символ $.

Если вы хотите присваивать значения элементам массива по одному, приведенное выше списочное присваивание можно записать так:

```
my @home;
$home[0] = "кушетка";
$home[1] = "стул";
$home[2] = "стол";
$home[3] = "печка";
```

Как видно из примера, переменные можно создавать с помощью директивы my, не присваивая начальное значение. (Для отдельных элементов массива директива my не используется, потому что массив уже существует и он сам знает, как создавать элементы по мере необходимости.)

Поскольку массивы упорядочены, над ними можно выполнять различные полезные действия, такие как стековые операции push и pop. Стек, в сущности, является просто упорядоченным списком с началом и концом. Что особенно важно, с концом. Perl рассматривает конец массива как вершину стека. (Хотя большинство программистов на Perl представляют себе массивы горизонтальными последовательностями, у которых вершина стека справа.)

Хеши. *Хеш* представляет собой неупорядоченный набор скаляров, к которым обращаются по некоторому строковому значению[1], связанному с каждым скаляром. По этой причине хеши часто называют *ассоциативными массивами*. Но это слишком длинное название для ввода с клавиатуры, а упоминаются ассоциативные массивы столь часто, что мы решили дать им какое-нибудь краткое и броское название. Другой причиной, по которой мы выбрали название «хеш», было желание подчеркнуть тот факт, что подобные массивы не упорядочены. (Так совпало, что в их внутренней реализации используется поиск по хеш-таблице, в результате чего работа с ними происходит столь быстро вне зависимости от числа хранимых значений.) Однако применить к хешу операцию push или pop нельзя: это не имеет смысла. У хеша нет начала и конца. Тем не менее хеши – это невероятно мощный и полезный инструмент. Пока вы не начали думать на языке хешей, вы, в сущности, не начали думать на Perl. На рис. 1.1 показаны упорядоченные элементы массива и неупорядоченные (но поименованные) элементы хеша.

Поскольку ключи для хеша не задаются автоматически порядком значений, при заполнении хеша необходимо задавать как значение, так и ключ. Можно присваивать хешу список, так же как и массиву, но при этом каждая *пара* элементов списка будет интерпретироваться как ключ и значение. Поскольку мы работаем с парами элементов, имена хешей помечаются разыменовывающим префиксом %. (Если внимательно посмотреть на символ %, можно заметить ключ и значение, разделенные косой чертой. Если не получается, попробуйте прищуриться.)

---

[1] Или по ключу (порядковому номеру, нижнему индексу), определяющему местоположение. Выберите вариант по душе.

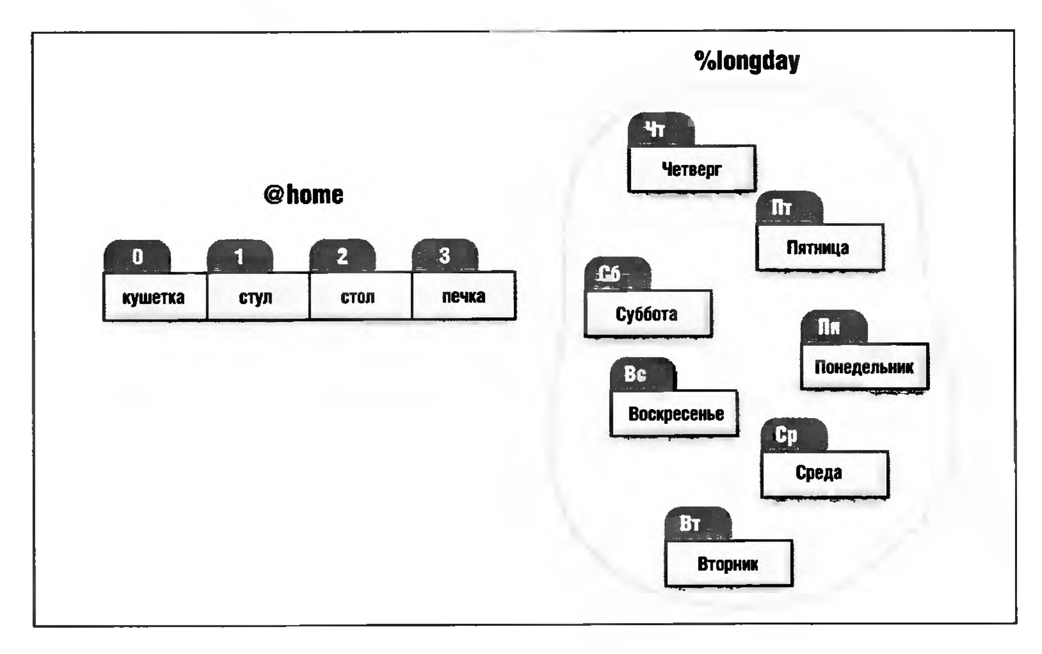


***Рис. 1.1.** Массив и хеш*

Предположим, вам требуется переводить сокращенные названия дней в соответствующие полные названия. Можно создать такое списочное присваивание:

```
my %longday = ("Вс", "Воскресенье", "Пн", "Понедельник", "Вт", "Вторник",
               "Ср", "Среда", "Чт", "Четверг", "Пт",
               "Пятница", "Сб", "Суббота");
```

Читать такое довольно трудно, поэтому Perl предоставляет последовательность символов => (знак «равно», знак «больше») в качестве разделителя, альтернативного запятой. С использованием этой синтаксической поблажки (и творческого форматирования) становится значительно легче различать, какие строки являются ключами, а какие – связанными с ними значениями:

```
my %longday = (
    "Вс" => "Воскресенье",
    "Пн" => "Понедельник",
    "Вт" => "Вторник",
    "Ср" => "Среда",
    "Чт" => "Четверг",
    "Пт" => "Пятница",
    "Сб" => "Суббота",
);
```

Выше мы видели, что возможно присвоение списка хешу, но верно и обратное: хеш, упомянутый в списочном контексте, преобразуется обратно в список пар «ключ/значение», правда, в причудливом порядке. Иногда бывает полезно. Чаще требуется извлечь из хеша список одних ключей, для чего применяется функция с уместным названием keys. Список ключей тоже не упорядочен, но при желании его легко можно отсортировать с помощью столь же уместно названной функции

sort. После этого упорядоченные ключи можно использовать для извлечения соответствующих значений в требуемом порядке.

Поскольку хеш – довольно причудливая разновидность массива, доступ к отдельному элементу хеша осуществляется путем помещения ключа в фигурные скобки (причудливые скобки, которые известны как «завитушки»[1]). Поэтому если, скажем, требуется найти в вышеприведенном хеше значение, ассоциированное со средой (Ср), следует использовать $longday{"Ср"}. Еще раз обратите внимание, что мы имеем дело со скалярной величиной, поэтому идентификатор начинается символом $, а не %; последний указывал бы на весь хеш целиком.

Если смотреть с позиций лингвистики, закодированная в хеше связь аналогична родительному падежу или притяжательному местоимению, как слово «of» в английском языке или как «'s». Женой Адама является Ева, поэтому мы пишем:

```
my %wife;
$wife{"Адам"} = "Ева";
```

Сложности

Массивы и хеши являют собой прелестные, простые, плоские структуры данных. К несчастью, окружающий мир не всегда идет навстречу нашим попыткам чрезмерного упрощения. Иногда приходится создавать не столь прелестные, простые и плоские структуры данных. Perl допускает это, «делая вид», что сложные величины на самом деле являются простыми. Иначе говоря, Perl позволяет манипулировать простыми скалярными ссылками, которые указывают на сложные массивы и хеши. В естественном языке мы занимаемся этим постоянно, когда используем простое существительное в единственном числе, например, «правительство» для обозначения объекта, который исключительно запутан и непостижим. Помимо всего прочего.

Расширим наш предыдущий пример, предположив, что вместо жены Адама мы хотим поговорить о жене Иакова. Известно, между тем, что у Иакова было четыре жены. (Не пытайтесь повторить этот трюк.) Пытаясь представить это на Perl, мы оказываемся в необычном положении: требуется притвориться, что четыре жены Иакова в действительности являются одной женой. (Этот трюк тоже повторять не стоит.) Читатель может решить, что допустима такая запись:

```
$wife{"Иаков"} = ("Лия", "Рахиль", "Валла", "Зелфа") # НЕВЕРНО
```

Но в результате мы не получим того, что требуется, потому что в Perl даже скобки и запятые бессильны превратить список в скаляр. (Скобки предназначены для синтаксического объединения, а запятые – для синтаксического разделения.) Вместо этого необходимо явно сообщить Perl, что список требуется представить в виде скаляра. Оказывается, что достаточной силой для этого обладают квадратные скобки:

```
$wife{"Иаков"} = ["Лия", "Рахиль", "Валла", Зелфа"] # ok
```

Эта команда создает анонимный массив и помещает ссылку на него в элемент хеша $wife{"Иаков"}. В результате получаем именованный хеш, содержащий анонимный массив. Таким способом Perl работает с многомерными массивами и вло-

---

[1] Разумеется, речь об английском языке, где для обозначения фигурных скобок существуют слова brace(s), curly brace(s)/curly brackets или просто curlies (завитушки, кудряшки). – *Прим. ред.*

женными структурами данных. Как и в обычных массивах и хешах, можно присваивать значения отдельным элементам следующим образом:

```
$wife{"Иаков"}[0] = "Лия";
$wife{"Иаков"}[1] = "Рахиль";
$wife{"Иаков"}[2] = "Валла";
$wife{"Иаков"}[3] = "Зелфа";
```

Как видите, это похоже на многомерный массив, одним индексом которого является строка, а другим – число. Чтобы показать нечто более похожее по структуре на дерево, типа вложенных структур данных, предположим, что мы хотим перечислить не только жен Иакова, но и всех сыновей каждой из его жен. В этом случае надо обращаться с хешем как со скаляром. Для этого можно использовать скобки. (Внутри каждого значения хеша мы, как и раньше, будем применять для представления массивов квадратные скобки. Но теперь у нас массив, содержащийся в хеше, содержащемся в хеше.)

```
My %kids_of_wife,
$kids_of_wife{"Иаков"} = {
    "Лия" => ["Рувим", "Симеон", "Левий", "Иуда", "Иссахар", "Завулон"],
    "Рахиль" => ["Иосиф", "Вениамин"]
    "Валла" => ["Дан", "Неффалим"],
    "Зелфа" => ["Гад", "Асир"],
}
```

Это более или менее эквивалентно следующему:

```
my %kids_of_wife,
$kids_of_wife{"Иаков"}{"Лия"}[0]    = "Рувим";
$kids_of_wife{"Иаков"}{"Лия"}[1]    = "Симеон";
$kids_of_wife{"Иаков"}{"Лия"}[2]    = "Левий",
$kids_of_wife{"Иаков"}{"Лия"}[3]    = "Иуда";
$kids_of_wife{"Иаков"}{"Лия"}[4]    = "Иссахар";
$kids_of_wife{"Иаков"}{"Лия"}[5]    = "Завулон";
$kids_of_wife{"Иаков"}{"Рахиль"}[0] = "Иосиф",
$kids_of_wife{"Иаков"}{"Рахиль"}[1] = "Вениамин;
$kids_of_wife{"Иаков"}{"Валла"}[0]  = "Дан";
$kids_of_wife{"Иаков"}{"Валла"}[1]  = "Неффалим";
$kids_of_wife{"Иаков"}{"Зелфа"}[0]  = "Гад";
$kids_of_wife{"Иаков"}{"Зелфа"}[1]  = "Асир";
```

Как видите, добавление нового уровня во вложенную структуру данных похоже на добавление еще одного измерения в многомерный массив. Perl позволяет программисту применять любую умозрительную модель, при этом внутреннее представление не меняется.

Важный момент: Perl допускает считать сложную структуру данных простым скаляром. На этой простой разновидности инкапсуляции основывается вся объектно-ориентированная структура Perl. Когда выше мы вызвали конструктор для `Camel` в виде:

```
my $fido = Camel->new("Amelia")
```

то создали объект `Camel`, представленный скаляром `$fido`. Но внутреннее строение `Camel` является более сложным. Как воспитанные объектно-ориентированные программисты, мы не должны интересоваться содержимым объектов `Camel` (если

только перед нами не поставлена задача реализации методов класса Camel). Но обычно объект типа Camel состоит из хеша, содержащего атрибуты конкретного объекта Camel, такие как кличка (в данном случае "Amelia", а не "fido") и количество горбов (которое мы не задали, но по умолчанию, вероятно, установлен один, как на обложке этой книги).

Простые вещи

Если у вас не закружилась слегка голова после чтения предыдущего параграфа, ваша голова необычная. Как правило, люди не любят иметь дело со сложными структурами данных, будь они правительственными или генеалогическими. Поэтому в естественных языках есть много способов сделать вид, что никаких сложностей нет. Многие из них связаны с *заданием темы* (topicalization), причудливым лингвистическим термином, означающим соглашение с кем-то относительно того, о чем пойдет речь (или путем исключения того, о чем речь, вероятно, не пойдет). В языке это происходит на нескольких уровнях. На верхнем уровне мы разделяемся на различные субкультуры, интересующиеся различными подтемами и устанавливающие субъязыки, посвященные разговорам на такие подтемы. Жаргон медицинской клиники («нерастворимый фактор асфиксии» – indissoluble asphyxiant) отличен от жаргона кондитерской фабрики («долгоиграющий гобстоппер» – everlasting gobstopper[1]). В основном мы автоматически переключаем контексты при переходе с одного жаргона на другой.

На разговорном уровне переключение контекста должно производиться более явным образом, поэтому в нашем языке есть много способов сообщить, о чем мы собираемся говорить. Мы озаглавливаем книги и разделы, из которых они состоят. В свои предложения мы вставляем витиеватые фразы, такие как «касательно вашего последнего запроса» или «для каждого X». Обычно, однако, мы просто произносим нечто вроде: «Ну, знаешь, эта висюлька, которая болтается у тебя в задней части горла?»

Perl также предлагает несколько способов задать тему. Одним из важных способов является объявление пакета, package. Предположим, что мы хотим поговорить на Perl о верблюдах. Скорее всего, следует начать модуль Camel словами:

```
package Camel;
```

У этого объявления несколько примечательных последствий. В частности, начиная с этого места, Perl будет считать все неопределенные глаголы и существительные относящимися к Camel. Он делает это, автоматически предваряя все глобальные имена префиксом Camel::[2]. Поэтому если сказать:

```
package Camel;
our $fido = &fetch();
```

[1] Отличительной особенностью этих кондитерских изделий является их твердость; размеры и ингредиенты могут быть самыми разными. – *Прим. ред.*

[2] Объявлять глобальные переменные можно с помощью декларации our, которая очень похожа на my, но означает, что переменная является разделяемой, или совместно используемой. Переменные my не могут использоваться совместно и недоступны за пределами текущего блока. Если сомневаетесь, используйте объявление my вместо our, поскольку лишние глобальные имена только захламляют наш мир и приводят других в замешательство.

то настоящим именем $fido будет $Camel::fido (а настоящим именем &fetch будет &Camel::fetch, но пока мы не будем говорить о глаголах). Это означает, что если в другом модуле сказано:

```
package Dog;
our $fido = &fetch();
```

то Perl не напутает, поскольку настоящим именем этой $fido будет $Dog::fido, а не $Camel::fido. «По-научному» говорится, что пакет устанавливает *пространство имен* (namespace). Количество пространств имен не ограничено, но поскольку одновременно нельзя находиться сразу в нескольких, то можно притвориться, что остальных пространств имен не существует. Благодаря этому пространства имен упрощают жизнь. Упрощение основывается на притворстве. (Так же, конечно, как и чрезмерное упрощение, которым мы занимаемся в данной главе.)

Содержать в порядке существительные важно, но так же важно содержать в порядке глаголы. Прекрасно, что &Camel::fetch нельзя спутать с &Dog::fetch в пространствах имен Camel и Dog, но действительно замечательно в пакетах то, что они классифицируют глаголы так, что ими могут пользоваться *другие* пакеты. Когда мы говорим:

```
my $fido = Camel->new("Amelia");
```

то фактически применяем глагол &new из пакета Camel, и полным именем этого глагола является &Camel::new. А когда мы говорим:

```
$fido->saddle();
```

то вызываем метод &Camel::saddle, поскольку переменная $fido помнит, что она указывает на Camel. Вот так устроено объектно-ориентированное программирование.

Словами package Camel мы начинаем новый пакет. Но иногда требуется взять существительные и глаголы из уже существующего пакета. Perl позволяет сделать это посредством объявления use (использовать), которое обеспечивает не только заимствование глаголов из другого пакета, но и загрузку указанного модуля с диска. На деле, *необходимо* сказать что-то вроде:

```
use Camel,
```

перед тем как сказать:

```
my $fido = Camel->("Amelia");
```

потому что иначе Perl не будет знать, что такое Camel.

Интересно, что разработчику, в сущности, не требуется знать, что такое Camel, при условии, что кто-то другой может написать модуль Camel вместо него. Еще лучше, если кто-то *уже* написал модуль Camel. Можно утверждать, что мощь Perl заключается не в самом Perl, а в CPAN (Comprehensive Perl Archive Network; см. главу 19) – архиве, содержащем мириады модулей для самых разных задач, от решения которых вы освобождаетесь. Вам нужно лишь загрузить любой понравившийся модуль и сказать:

```
use Some::Cool::Module;
```

После этого можно использовать глаголы из этого модуля так, как это уместно в контексте разговора на определенную тему.

Подобно тому, как это происходит в естественном языке, установка темы в Perl «искажает» язык, который будет использоваться с данного места и до конца области видимости. На практике некоторые встроенные модули вообще не вводят в действие новые глаголы, а только деформируют язык Perl разными полезными способами. Эти особые модули мы называем *прагмами* (pragmas, см. главу 29). Например, часто применяется прагма `strict`:

```
use strict;
```

Действие модуля `strict` состоит в ужесточении некоторых правил, в результате чего чаще приходится указывать различные вещи, о которых в другом случае Perl догадывался бы сам, например, об областях видимости переменных[1]. Явные указания выгодны при работе над большими проектами. По умолчанию Perl оптимизирован для маленьких проектов, но в случае применения прагмы `strict` становится подходящим инструментом и для больших проектов, требующих лучшего сопровождения. Поскольку прагму `strict` можно добавить в любой момент, Perl хорошо справляется с разрастанием маленьких проектов в большие, даже если поначалу это не предполагалось. Как обычно и бывает.

Вместе с языком Perl развивается и его сообщество, и по мере развития изменяются представления сообщества о том, как должен вести себя Perl по умолчанию. (Что противоречит желанию, чтобы Perl вел себя так же, как прежде.) Так, например, большинство программистов на Perl теперь думают, что всегда следует вставлять `use strict` в начало программы. С течением времени происходит накопление таких «культурных» прагм, деформирующих язык. Поэтому была добавлена другая встроенная прагма, состоящая исключительно из номера версии Perl, являющаяся своеобразной «метапрагмой», сообщающей интерпретатору Perl, что он должен действовать как более современный язык:

```
use v5.14;
```

Данное конкретное объявление активизирует сразу несколько прагм, включая `use strict`;[2] оно также включает некоторые новые функции, такие как глагол `say`, который (в отличие от `print`) автоматически добавляет перевод строки. Соответственно, наш самый первый пример можно записать так:

```
use v5.14;
say "Howdy, world!";
```

Все примеры в этой книге написаны в предположении, что используется версия Perl v5.14; мы постараемся не забывать включать объявление `use v5.14` в листинги полных программ, но при демонстрации фрагментов мы будем исходить из того, что вы уже добавили это объявление самостоятельно. (Если вы пользуетесь не последней версией Perl, некоторые из наших примеров могут оказаться неработоспособными. В случае с командой `say` просто замените ее командой `print`, добавив сим-

---

[1] В частности, инструкция `use strict` требует, чтобы вы использовали директивы `my`, `state` или `our` в объявлениях переменных, иначе будет предполагаться, что необъявленные переменные являются переменными пакета, что может привести к неприятностям. Она также запрещает использование различных конструкций, за долгие годы существования которых выяснилось, что они чреваты ошибками.

[2] Неявная поддержка прагмы `strict` была добавлена в версии v5.12. Обратите также внимание на прагму `feature`, описываемую в главе 29.

вол перевода строки, но лучше будет обновить версию Perl. Чтобы использование команды say не вызывало проблем, вы должны сказать как минимум use v5.10.)

Глаголы

Как это свойственно обычным «повелительным» языкам программирования, многие глаголы Perl являются командами: они приказывают интерпретатору Perl что-либо выполнить. С другой стороны, что характерно для естественных языков, значения глаголов Perl имеют тенденцию «разбегаться» в разных направлениях в зависимости от контекста. Высказывание, начинающееся с глагола, обычно является чисто повелительным и выполняется исключительно ради своих побочных эффектов. (Иногда мы называем такие глаголы *процедурами* (*procedures*), особенно если они определяются пользователем.) К часто встречающимся встроенным командам относится команда say (мы с ней уже знакомы):

```
say "Женой Адама является $wife{'Адам'}. ";
```

Ее побочный эффект выражается в появлении желаемого вывода:

```
Женой Адама является Ева.
```

Но кроме повелительного существуют и другие «наклонения». Некоторые глаголы применяются в вопросительных фразах и полезны в условных операторах, таких как оператор if. Другие глаголы транслируют свои входные параметры в возвращаемые значения, подобно тому как рецепт описывает способ превращения исходных ингредиентов в нечто (как можно надеяться) съедобное. Мы склонны называть такие глаголы *функциями* (*functions*) из уважения к поколениям математиков, которым неизвестно значение слова «функциональный» в повседневном языке.

Примером встроенной функции служит функция экспоненты:

```
my $e = exp(1), # 2 718281828459 или около того
```

Однако в Perl не проводится жесткого разделения между процедурами и функциями. Читатель обнаружит, что эти термины используются взаимозаменяемым образом. Иногда глаголы называются также операторами (если они встроенные) или подпрограммами (если они определены пользователем).[1] Называйте их как угодно, но все они возвращают значение, которое может быть осмысленным или нет и которое можно по необходимости игнорировать.

По ходу нашего изложения вам встретятся и другие примеры того, как Perl ведет себя аналогично естественным языкам. Но можно взглянуть на Perl и с других

---

[1] Исторически сложилось так, что Perl требовал указания символа «амперсанда» (&) при любых вызовах подпрограмм, определенных пользователем (смотри выше $fido = &fetch();). Но в Perl версии 5 использование амперсанда стало необязательным, поэтому при обращении к глаголам, определенным пользователем, теперь допускается такой же синтаксис, как и со встроенными глаголами ($fido = fetch();). Мы по-прежнему используем амперсанд, говоря об *имени* подпрограммы, например, при получении ссылки на нее ($fetcher = \&fetch;). С позиций лингвистики можно рассматривать формат &fetch с амперсандом как инфинитив, «to fetch», или аналогичную форму «do fetch». Но мы редко говорим «do fetch», если можно сказать просто «fetch». Это и есть действительная причина того, что мы отбросили обязательный амперсанд в Perl 5.

точек зрения. Мы уже «втихомолку» ввели некоторые обозначения из языка математики, такие как индексы, сложение, функция экспоненты. Но Perl является также языком управления, связующим языком (glue language), языком создания прототипов, языком обработки текста, обработки списков и объектно-ориентированным языком – помимо всего прочего.

Perl – это также старый добрый язык программирования для компьютеров. С этой точки зрения мы и рассмотрим его теперь.

Пример вычисления среднего

Предположим, что вы преподаете Perl группе студентов и пытаетесь решить, как выставлять оценки. У вас есть результаты сдачи экзаменов каждым студентом, причем результаты идут в произвольном порядке. Требуется получить список, объединяющий оценки каждого студента и дополненный средним баллом. И у вас есть текстовый файл (оригинально названный *grades* – оценки), который выглядит так:

```
Noel 25
Ben 76
Clementine 49
Norm 66
Chris 92
Doug 42
Carol 25
Ben 12
Clementine 0
Norm 66
...
```

С помощью следующего сценария можно собрать вместе все оценки, определить средний балл для каждого студента и вывести все это в алфавитном порядке. В программе наивно предполагается, что в группе нет двух студенток с именем Carol. А именно, если есть запись, относящаяся ко второй Carol, программа решит, что это еще одна оценка для первой Carol (не путать с первой Noël)

Кстати, номера строк не являются частью программы, несмотря на возможное сходство с BASIC в других аспектах.

```
 1 #!/usr/bin/perl
 2 use v5.14;
 3
 4 open(GRADES, "<:utf8", "grades") or die "Невозможно открыть grades: $!\n";
 5 binmode(STDOUT, ":utf8")
 6
 7 my %grades;
 8 while (my $line = <GRADES>) {
 9     my ($student, $grade) = split(" ", $line);
10     $grades{$student} .= $grade . " ";
11 }
12
13 for my $student (sort keys %grades) {
14     my $scores = 0;
15     my $total = 0;
16     my @grades = split(" ", $grades{$student});
```

```
    for my $grade (@grades) {
        $total += $grade;
        $scores++
    }
    my $average = $total / $scores;
    print "$student: $grades{$student}\tСреднее: $average\n";
}
```

Пока у вас не разбежались глаза, мы лучше сразу отметим, что в этом примере демонстрируется многое из того, о чем мы уже рассказывали, а еще много такого, что мы объясним вскоре. Но если взглянуть на этот пример несколько издалека, то можно обнаружить некоторые интересные закономерности. Попробуйте сами догадаться о том, что здесь происходит, а позднее мы сообщим, подтвердились ли эти догадки.

Мы бы посоветовали вам попробовать запустить этот пример на выполнение, но вы, вероятно, еще не знаете, как это делается.

Как это делается

Ох, сейчас вы, наверное, размышляете, как же запустить программу на Perl. Короткий ответ состоит в том, что нужно подать ее на вход интерпретатора языка Perl, который, по счастливому совпадению, носит название *perl*. Длинный ответ начинается так: «There's More Than One Way To Do It» – есть несколько способов сделать это.[1]

Первый способ запустить *perl* (как правило, работающий в любой операционной системе) – это просто вызвать *perl* явным образом из командной строки.[2] Если вы выполняете что-то очень простое, можно использовать ключ *-e* (в следующем примере символ % представляет стандартное приглашение оболочки, поэтому его набирать не нужно). В UNIX, к примеру, можно ввести:

```
% perl -e 'print "Hello, world!\n";'
```

В других операционных системах могут понадобиться некоторые махинации с кавычками. Но базовый принцип везде одинаковый: попытаться уместить все, что требуется сообщить Perl, примерно в 80 знаков.[3]

Для создания более длинных сценариев можно с помощью своего любимого текстового редактора (или любого другого текстового редактора) поместить все команды в файл, а затем, предполагая, что вы назвали этот файл *gradation*, сказать:

```
% perl gradation
```

---

[1] Это девиз Perl, и вам неоднократно придется слышать его, если только вы не являетесь местным экспертом в Perl, и тогда вам неоднократно придется его произносить. Иногда его сокращают до TMTOWTDI, что произносится «Тим Тоуди». Но каждый волен произносить это так, как ему больше нравится. В конце концов, TMTOWTDI.

[2] В предположении, что ваша система имеет интерфейс командной строки. Если у вас его нет, пора сделать апгрейд.

[3] Такого типа сценарии часто называют «однострочниками» («one-liners»). Попав в компанию других программистов на Perl, вы обнаружите, что некоторые из них увлекаются созданием замысловатых однострочников. Из-за этих махинаций злые языки иногда называют Perl языком «только для записи» (write-only language).

При этом интерпретатор Perl по-прежнему вызывается явным образом, но, по крайней мере, не приходится каждый раз вводить все в командной строке и не нужно возиться с кавычками, чтобы удовлетворить требованиям оболочки.

Удобнее всего вызвать сценарий, просто указав его имя (или щелкнув на нем) и предоставив операционной системе найти требуемый интерпретатор. В некоторых системах существуют способы связывания определенных расширений файлов или каталогов с теми или иными приложениями. В этих системах нужно произвести действия, необходимые для связывания сценария Perl с интерпретатором *perl*. В системах UNIX, поддерживающих нотацию #! (нотация «shebang»), а таковыми в наше время является большинство систем UNIX, можно с помощью специальной первой строки сценария указать операционной системе, какую программу она должна запустить. Введите в качестве первой строки нашего примера нечто аналогичное

```
#!/usr/bin/perl
```

(Если *perl* v5.14 находится не в */usr/bin*, то нужно соответствующим образом изменить строку #!.[1]) После этого вам требуется лишь сказать:

```
% gradation
```

Разумеется, это не сработало, поскольку вы забыли сделать сценарий выполняемым файлом (см. страницу руководства *chmod*(1)) и добавить его в маршрут поиска PATH.[2] Если его нет в переменной PATH, необходимо указать полное имя файла, чтобы операционная система знала, где найти ваш сценарий, например так:

```
% /home/sharon/bin/gradation
```

Наконец, если вам настолько не везет, что у вас стоит древняя система UNIX, которая не поддерживает волшебную строку #!, или путь к вашему интерпретатору длиннее 32 символов (встроенный предел многих систем), вы можете обойти эти ограничения таким способом:

```
#!/bin/sh -- # perl, остановить цикл
eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
    if 0;
```

В некоторых операционных системах требуется какая-нибудь разновидность этого приема, чтобы справиться с */bin/csh*, *DCL*, *COMMAND.COM* или другим интерпретатором командной строки, устанавливаемым по умолчанию. Посоветуйтесь с Местным Экспертом.

На протяжении всей этой книги мы будем использовать просто #!/usr/bin/perl для представления всех этих обозначений, но теперь вы знаете, что под этим имеется в виду.

---

1 Если вы пользуетесь старой версией */usr/bin/perl*, можно скомпилировать новую версию и поместить ее в другой каталог, например */usr/local/bin*, но не забудьте при этом исправить путь в строке #!.

2 В большинстве случаев, после того как файлу назначены атрибуты исполнения (той же командой chmod +x), если выполнить gradation в том каталоге, где файл хранится, этот файл будет отправлен на исполнение. Иначе говоря, его необязательно помещать в каталог, упомянутый в переменной среды PATH. Второе, что тут следует отметить, – в Windows фокус с chmod вообще не работает (ввиду отсутствия chmod). Там следует сопоставить расширение (.pl или .perl) интерпретатору Perl. – *Прим. науч. ред.*

Случайный совет: когда будете писать пробный сценарий, не называйте его *test*. В системах UNIX имеется встроенная команда *test*, которая, скорее всего, и будет выполнена вместо вашего сценария. Попробуйте лучше назвать сценарий *try*.

Теперь, когда вы знаете, как запустить свою собственную программу на Perl (не путать с программой *perl*), вернемся к нашему примеру.

Дескрипторы файлов

Если только вы не занимаетесь моделированием философа-солипсиста с помощью методов искусственного интеллекта, вашей программе требуются какие-то средства общения с внешним миром. В строках 4 и 8 нашего Примера вычисления среднего встречается слово `GRADES`, которое иллюстрирует еще один из типов данных Perl, а именно *дескриптор файла* (*filehandle*). Дескриптор файла – это просто имя, которое присваивается файлу, устройству, сокету или конвейеру и помогает вспомнить, о чем из перечисленного вы говорите, а также скрыть некоторые сложные вещи типа буферизации. (Во внутренней реализации дескрипторы файлов аналогичны потокам (streams) языков типа C++ или каналам ввода/вывода в BASIC.)

Дескрипторы файлов облегчают получение входных данных и отправку выходных в различные места. Хорошим связующим языком (glue language) Perl делает отчасти то, что он умеет одновременно общаться со многими файлами и процессами. Наличие удобных символических имен для различных внешних объектов – один из элементов хорошего связующего языка.[1]

Создание дескриптора файла и закрепление его за файлом выполняется с помощью функции `open`. Функция `open` принимает по меньшей мере два параметра: дескриптор файла и имя файла, с которым его нужно связать. Perl предоставляет также несколько предварительно определенных (и предварительно открытых) дескрипторов файлов. `STDIN` – это обычный канал ввода для вашей программы, а `STDOUT` – это для нее обычный канал вывода. `STDERR` представляет собой дополнительный канал вывода, позволяющий программе отпускать ехидные ремарки, пока она преобразует (или пытается преобразовать) ваши входные данные в выходные.[2] В строках 4 и 5 нашей программы мы также сообщили нашему новому дескрипто-

1 Вот еще аспекты, благодаря которым Perl является хорошим связующим языком: он способен обрабатывать не только символы ASCII, его можно встраивать, а с помощью модулей расширения в него можно встраивать другие компоненты. Он лаконичен и легко интегрируется с другими платформами. Он, можно сказать, ответственно подходит к вопросам экологии. Он может вызываться различными способами (как было показано выше). Но главное, сам язык не настолько строго структурирован, чтобы нельзя было каким-то способом решить стоящую перед вами проблему. Здесь снова проявляется уже знакомый нам принцип TMTOWTDI.

2 Обычно эти дескрипторы файлов привязаны к терминалу, чтобы можно было ввести данные с клавиатуры и посмотреть результат на экране, но можно связать их и с файлами (или чем-то еще). Perl предлагает эти предопределенные дескрипторы, поскольку они уже тем или иным способом созданы операционной системой. В UNIX процессы наследуют стандартные устройства ввода, вывода и вывода ошибок от родительского процесса, которым обычно является оболочка. Одной из задач оболочки является настройка этих потоков ввода/вывода таким образом, чтобы порожденному процессу не пришлось о них беспокоиться.

ру GRADES и существующему дескриптору STDOUT, что текст будет представлен в кодировке UTF-8, которая обычно используется для представления текста Юникода.

Поскольку посредством функции open можно открывать дескрипторы файлов в различных целях (для ввода, вывода, конвейеризации), необходимо иметь возможность указать требуемый режим. Аналогично тому, как это делается в командной строке, нужно просто добавить символы к имени файла.

```
open(SESAME, "имя_файла")         # чтение существующего файла
open(SESAME, "< имя_файла")       # (то же, но явным образом)
open(SESAME, "> имя_файла")       # создать файл и производить в него запись
open(SESAME, ">> имя_файла")                # дописывание в конец файла
open(SESAME, "| выходная_команда_конвейера")  # организовать выходной фильтр
open(SESAME, "входная_команда_конвейера |")   # организовать входной фильтр
```

Однако рекомендуемая к использованию версия функции open, принимающая три аргумента, позволяет указать режим использования файла в отдельном аргументе. Это удобно, когда приходится иметь дело с именами файлов, которые не являются литералами и могут содержать символы, напоминающие символы режима, или значимые пробелы.

```
open(SESAME, "<", $somefile)      # чтение существующего файла
open(SESAME, ">", $somefile)      # создать файл и производить в него запись
open(SESAME, ">>", $somefile)     # дописывание в конец имеющегося файла
open(SESAME, "|-", "выходная_команда_конвейера")  # организовать выходной фильтр
open(SESAME, "-|", "входная_команда_конвейера")   # организовать входной фильтр
```

Эта версия функции open также позволяет указать кодировку символов в файле, что мы и проделали в нашей программе-примере.

```
open(SESAME, "< :encoding(UTF-8)",     $somefile)
open(SESAME, "> :crlf",                $somefile)
open(SESAME, ">> :encoding(MacRoman)", $somefile)
```

Как можно видеть, имя для дескриптора файла выбирается произвольно. После открытия дескриптор файла SESAME можно использовать для доступа к файлу или конвейеру, пока он не будет явным образом закрыт (с помощью, как можно догадаться, close(SESAME)) либо пока дескриптор файла не будет связан с новым файлом посредством нового вызова open с тем же самым дескриптором в качестве первого аргумента. Повторное открытие уже открытого дескриптора файла приводит к неявному закрытию первоначального файла, который становится недоступным дескриптору, и открытию нового файла. Необходимо проявлять осторожность – действительно ли это то, что вы собираетесь делать? Иногда это происходит случайно, например, если вы сказали open($handle,$file), а $handle содержит ту же строку, что и раньше. Следите, чтобы переменная $handle указывала на уникальную строку, иначе будет открыт новый файл с прежним дескриптором.

Но лучше оставить $handle неопределенной – Perl самостоятельно выберет для нее значение. Этот прием может пригодиться, когда вы устанете выбирать имена для своих дескрипторов: если передать функции open переменную с неопределенным значением (например, созданную с помощью объявления my), Perl автоматически выберет строку для обозначения дескриптора файла и определит переменную:

```
open(my $handle, "< :crlf :encoding(cp1252)", $somefile)
    || die "невозможно открыть $somefile. $!";
```

Если вызов функции open увенчался успехом, переменная $handle определена, и ее можно использовать в качестве дескриптора файла.

Открыв дескриптор файла для ввода, можно прочесть строку с помощью оператора чтения строки, <>. Его называют также оператором угловых скобок (angle operator). В оператор угловых скобок заключается имя дескриптора файла (<SESAME>, если речь идет о дескрипторе-литерале, или <$handle> в случае косвенного дескриптора), строки которого требуется читать. Пустой оператор угловых скобок <> читает строки из всех файлов, указанных в командной строке, или из STDIN, если файлы не указаны. (Это стандартный режим для многих программ-фильтров.) Пример использования дескриптора файла STDIN для чтения данных, вводимых пользователем, может выглядеть примерно так:

```
print STDOUT "Введите число: "          # запрос числа
$number = <STDIN>;                      # ввод числа
say STDOUT "Вы ввели число $number.\n"; # вывод числа
```

Заметили, что мы только что вам подкинули? Что там делает STDOUT в этих операторах print и say? Это всего лишь один из способов использования дескриптора файла выходных данных. Дескриптор файла можно передавать между оператором и списком его аргументов, и если дескриптор указан, то определяет, куда должна быть направлена выдача. В данном случае указание дескриптора файла излишне, поскольку и без него выдача была бы направлена на STDOUT. Совсем как STDIN устанавливается по умолчанию для ввода, STDOUT устанавливается по умолчанию для вывода. (В строке 22 нашего Примера вычисления среднего мы опустили дескриптор, чтобы не привести вас раньше времени в замешательство.)

Запустив предыдущий пример, можно заметить, что выводится лишняя пустая строка. Это происходит потому, что операция чтения строки не удаляет автоматически символ перевода строки из строки ввода (пользователь мог ввести, например, "9\n"). Для тех случаев, когда нужно удалить символ перевода строки, в Perl есть функции chop и chomp. Функция chop удаляет любой последний символ строки и возвращает его, в то время как chomp удаляет только маркер конца записи (обычно "\n") и возвращает число удаленных символов. Часто можно видеть такую идиому ввода отдельной строки:

```
chomp($number = <STDIN>), # ввести число и удалить перевод строки
```

что равносильно следующему:

```
$number = <STDIN>;          # ввести число
chomp($number);             # удалить перевод строки
```

И последнее замечание: не стоит считать значение переменной числом только потому, что она называется $number. В ней может храниться любая строка. Perl попытается интерпретировать ее как число, только если вы используете ее в численном выражении, а это – тема следующего раздела, посвященного операторам.

Операторы

Как мы намекали ранее, Perl является и математическим языком. Это справедливо на многих уровнях, начиная с низкоуровневых поразрядных логических операций, включая операции над числами и множествами и далее, вплоть до более крупных предикатов и абстракций различного типа. А как известно из школьного

курса математики, математикам очень по душе необычные символы. Хуже того, специалисты по вычислительной технике придумали свои варианты этих странных символов. В Perl тоже есть несколько таких странных символов, но, по правде сказать, в большинстве своем они непосредственно заимствованы из C, FORTRAN, *sed*(1) или *awk*(1), поэтому пользователям этих языков они должны быть знакомы.

Остальные могут утешиться тем, что изучение странных символов в Perl может послужить началом изучения всех этих странных языков.

Встроенные операторы Perl могут быть классифицированы по числу операндов как унарные (одноместные), бинарные (двухместные) и тернарные (трехместные) операторы. Их можно классифицировать по тому, являются ли они префиксными (предшествующими своим операндам) или инфиксными («разбавляющими» свои операнды). Можно также классифицировать их по типу объектов, с которыми они работают, например, с числами, строками, файлами. Далее мы приведем таблицу со всеми операторами, но для начала рассмотрим несколько полезных операторов.

Некоторые бинарные арифметические операторы

Арифметические операторы делают именно то, что можно подумать, если вы встречали их в школе. Они выполняют некоторые математические функции над числами (табл. 1.2).

Таблица 1.2. Математические операторы

Пример	Название	Результат
`$a + $b`	Сложение	Сумма `$a` и `$b`
`$a * $b`	Умножение	Произведение `$a` и `$b`
`$a % $b`	Взятие по модулю	Остаток от деления `$a` на `$b`
`$a ** $b`	Возведение в степень	`$a` в степени `$b`

Да, мы опустили вычитание и деление – полагаем, что читатель сообразит, как они действуют. Поработайте с ними и проверьте, правы ли вы. (Или смошенничайте и посмотрите в главе 3.) Арифметические операторы выполняются в том порядке, который вам преподал школьный учитель математики (возведение в степень раньше умножения; умножение раньше сложения). Для изменения очередности всегда можно воспользоваться скобками.

Строковые операторы

Существует также оператор «сложения» для строк, осуществляющий конкатенацию, т.е. соединяющий строки. В отличие от языков, в которых эта операция выглядит так же, как сложение чисел, в Perl для конкатенации строк определен отдельный оператор (.):

```
$a = 123;
$b = 456;
say $a + $b; # выводит 579
say $a . $b; # выводит 123456
```

Есть также оператор «умножения» для строк, называемый оператором *повторения* (repeat). Опять же, это самостоятельный оператор (x), отличный от оператора умножения чисел:

```
$a = 123;
$b = 3;
say $a * $b; # выводит 369
say $a x $b; # выводит 123123123
```

Эти строковые операторы связывают значения с тем же приоритетом, что и соответствующие им арифметические операторы.[1] Оператор повторения несколько необычен тем, что принимает строку в качестве левого аргумента и число в качестве правого. Обратите также внимание, что Perl автоматически преобразует числа в строки. Все приведенные выше числа-литералы можно было бы заключить в кавычки, и результат от этого не изменился бы. Внутренне, однако, преобразование производилось бы в обратном направлении, т.е. из строк в числа.

Стоит обратить внимание еще на некоторые вещи. Конкатенация строк выполняется также при интерполяции, происходящей в строках, заключенных в двойные кавычки. И при выводе списка значений фактически производится конкатенация строк. В результате три следующие команды дают одинаковый вывод:

```
say $a . " равно " . $b . ".";  # оператор "точка"
say $a, " равно ", $b, ".";     # список
say "$a равно $b.";             # интерполяция
```

Которую из них использовать в конкретной ситуации, зависит целиком от вас. (Однако мы считаем, что интерполяцию обычно легче всего читать.)

Оператор x на первый взгляд кажется относительно бесполезным, но иногда он очень удобен, особенно для таких вещей:

```
say "-" x $scrwid;
```

В результате поперек экрана вычерчивается линия, если, конечно, $scrwid содержит число, соответствующее ширине экрана в символах, а не идентификатор винта (screw identifier).

Операторы присваивания

Хотя этот оператор является не совсем математическим, мы уже широко пользовались простым оператором присваивания =. Постарайтесь запомнить, что = означает «устанавливается в значение», а не «равно». (Существует и математический оператор проверки равенства ==, который означает «равно», и если вы с самого начала усвоите разницу между ними, то убережетесь от возникновения проблем в будущем. Оператор == действует как функция, возвращающая логическое значение, тогда как = больше похож на процедуру, которая выполняется для получения побочного эффекта модификации переменной.)

Как и операторы, описанные выше, операторы присваивания представляют собой бинарные инфиксные операторы; это значит, что их операнды расположены

---

[1] В настоящей книге, а также в документации Perl авторы используют понятие «связывания» или «скрепления» (английский глагол bind) для образной передачи интерпретации языком приоритетов операторов. В сложных выражениях вроде $b + $c * $d оператор * «связывает» значения переменных $c и $d сильнее, чем оператор + связывает значения переменных $b и $c, поэтому умножение имеет больший приоритет и вычисляется перед сложением. В том же контексте могут использоваться словосочетания вроде «более сильное связывание». – *Прим. науч. ред.*

по обеим сторонам оператора. Правый операнд может быть произвольным выражением, но левый должен быть допустимым *l-значением*[1] (lvalue), что в переводе на обычный язык означает допустимую область памяти типа переменной или элемента массива. Чаще всего используется простой оператор присваивания. Он вычисляет значение выражения, находящегося в правой части, а затем устанавливает переменную в левой части равной этому значению:

```
$a = $b;
$a = $b + 5;
$a = $a * 3;
```

Заметьте, что последнее присваивание дважды ссылается на одну и ту же переменную: один раз при вычислении, и один раз при присваивании. В этом нет ошибки, но такая операция выполняется столь часто, что для нее существует сокращенное обозначение (заимствованное из С). Операция вида:

```
l-значение оператор= выражение
```

выполняется так, как если бы было записано:

```
l-значение = l-значение оператор выражение
```

за исключением того, что l-значение не вычисляется дважды. (Разница есть лишь, когда вычисление l-значения имеет побочные эффекты. Но в тех случаях, когда разница *есть*, она обычно состоит именно в том, что вам требуется. Так что не стоит об этом волноваться.)

Так, предыдущий пример можно было бы записать в следующем виде:

```
$a *= 3;
```

что следует читать как «умножить $a на 3». Такое сокращение допустимо в Perl почти для любого бинарного оператора, даже для некоторых операторов, с которыми в С это нельзя делать:

```
$line .= "\n";  # Дописать символ перевода строки к $line
$fill x= 80;    # Сделать строку $fill 80-кратным повторением самой себя.
$val ||= "2";   # Установить значение $val равным 2, если оно не "true"
```

В строке 10 нашего Примера вычисления среднего[2] есть две конкатенации строк, одна из которых представляет собой оператор присваивания. А строка 18 содержит сокращенный оператор +=.

Независимо от того, какого типа оператор присваивания используется, в качестве значения присваивания в целом возвращается конечное значение переменной в левой части.[3] Это не удивит программистов на С, которым известно, как использовать эту идиому для обнуления переменных:

---

[1] Термин lvalue (от left value – левое значение) берет начало от оператора присваивания E1 = E2, где левый аргумент должен быть l-значением. – *Прим. науч. ред.*

[2] А вы думали, что мы про него забыли?

[3] Это не так, как, скажем, в Pascal, где присваивание является оператором и не возвращает никакого значения. Мы уже говорили, что присваивание аналогично процедуре, но запомните, что в Perl даже процедуры возвращают значения.

```
$a = $b = $c = 0,
```

Также часто можно видеть, что присваивание используется в условии цикла while, как в строке 8 нашего Примера вычисления среднего.

А вот что *должно* удивить программистов на C, так это то, что присваивание в Perl возвращает фактическую переменную в виде l-значение, поэтому можно модифицировать одну и ту же переменную несколько раз в одном операторе. Например, можно написать:

```
($temp -= 32) *= 5/9.
```

для преобразования по месту из шкалы Фаренгейта в шкалу Цельсия. Это еще одна причина, по которой ранее в этой главе мы могли сказать:

```
chomp ($number = <STDIN>);
```

чтобы «обрубить» конечное значение $number. Вообще говоря, эту особенность можно использовать всякий раз, когда требуется что-то скопировать и одновременно произвести над этим значением какие-то действия.

Унарные арифметические операторы

Как если бы запись $variable += 1 была недостаточно короткой, Perl заимствует из C еще более лаконичный способ приращения переменной. Операторы инкремента и декремента просто увеличивают или уменьшают значение переменной на единицу. Они могут помещаться с любой стороны от переменной, в зависимости от того, в какой момент требуется выполнить операцию (табл. 1.3).

Таблица 1.3. Операторы инкремента

Пример	Название	Результат
++$a, $a++	Инкремент	Прибавить 1 к $a
--$a, $a--	Декремент	Вычесть 1 из $a

Когда один из этих «автоматических» операторов предшествует переменной, это называется префиксным инкрементированием (декрементированием) переменной, а значение переменной будет изменено перед тем, как произойдет обращение к ней. Когда оператор помещается после переменной, то это называется постфиксным инкрементированием (декрементированием) переменной, и ее значение изменяется после использования. Например:

```
$a = 5;     # $a получает значение 5
$b = ++$a;  # $b получает инкрементированное значение $a, т.е. 6
$c = $a--;  # $c присваивается 6, затем $a декрементируется и становится 5
```

В строке 19 нашего Примера вычисления среднего число результатов увеличивается на единицу, и мы получаем количество результатов, которые нужно усреднить. При этом используется оператор постфиксного инкрементирования ($scores++), но в данном случае это несущественно, поскольку выражение применяется в пустом контексте, что представляет собой лишь причудливый способ

сказать, что выражение вычисляется только ради побочного эффекта инкрементирования переменной. Возвращаемое значение при этом отбрасывается.[1]

Логические операторы

Логические операторы, которые называют также операторами «короткого пути» («short-circuit»), позволяют программе принимать решения, исходя из нескольких критериев, без применения вложенных операторов if. Их называют операторами короткого пути, поскольку они пропускают (закорачивают) вычисление правого аргумента, если считают, что в левом аргументе содержится достаточно информации для получения итогового результата. Это делается не только для повышения производительности. Программист может явным образом использовать этот режим закорачивания, чтобы избежать вычислений в правом аргументе, о которых ему известно, что они сорвались бы, если бы их не «охранял» левый аргумент. В Perl можно сказать: «California or bust!» (Калифорния или провал), и при этом не провалиться (предполагая, что вы действительно доберетесь до Калифорнии).

На самом деле в Perl есть два набора логических операторов: традиционный набор, заимствованный из C, и более новый (но еще более традиционный) набор операторов со сверхнизким приоритетом, заимствованный из BASIC. Оба набора при надлежащем применении повышают удобочитаемость программ. Операторы C, использующие знаки пунктуации, уместны, когда желательно, чтобы логические операторы имели больший приоритет, чем запятые, тогда как записываемые словами операторы BASIC хорошо использовать, когда желательно, чтобы запятые имели больший приоритет, чем логические операторы. Часто они, в конечном итоге, действуют одинаково, а использование того или иного набора является делом вкуса. (Примеры, выявляющие контраст, можно найти в разделе «Логические and, or, not и xor» главы 3.) Хотя из-за различных приоритетов эти два набора операторов не являются взаимозаменяемыми, после синтаксического анализа сами операторы ведут себя одинаково; приоритетность просто определяет область расположения их аргументов. В табл. 1.4 перечислены логические операторы.

Таблица 1.4. Логические операторы

Пример	Название	Результат
`$a && $b`	И	$a, если $a ложно, в противном случае $b
`$a || $b`	ИЛИ	$a, если $a истинно, в противном случае $b
`! $a`	НЕ	Истинно, если $a не истинно
`$a and $b`	И	$a, если $a ложно, в противном случае $b
`$a or $b`	ИЛИ	$a, если $a истинно, в противном случае $b
`not $a`	НЕ	Истинно, если $a не истинно
`$a xor $b`	ИСКЛЮЧАЮЩЕЕ ИЛИ	Истинно, если $a или $b истинны, но не оба сразу

[1] Оптимизатор обратит на это внимание и преобразует постфиксное инкрементирование в префиксное, поскольку последнее выполняется немного быстрее. (Вам необязательно знать об этом, но мы надеемся, что это вас ободрит.)

Благодаря тому, что логические операторы «закорачивают» вычисление, их часто используют в Perl для условного выполнения кода. В следующей строке (строка 4 нашего Стандартного примера) делается попытка открыть файл *grades*:

```
open(GRADES, "<:utf8", "grades")  || die "Невозможно открыть grades: $!\n"
```

Если файл удается открыть, интерпретатор переходит к следующей строке программы. Если файл открыть не удается, выводится сообщение об ошибке и выполнение прекращается.

Буквально эта строка означает: «Открыть *grades* или умереть!» Операторы короткого пути служат еще одним примером естественности языка, но, кроме того, сохраняют наглядность исходного текста. Важные действия перечисляются в левой части экрана, а второстепенные скрыты справа. (Переменная `$!` содержит сообщение об ошибке, полученное от операционной системы – см. главу 25.) Конечно, эти логические операторы можно использовать и в более традиционных условных выражениях, таких как операторы `if` и `while`.

Некоторые операторы сравнения чисел и строк

Операторы сравнения, или отношения, сообщают нам о том, как две скалярные величины (числа или строки) относятся одна к другой. Есть два набора операторов: один сравнивает числа, а другой – строки. (В любом случае аргументам будет сначала «принудительно» навязан соответствующий тип.) Взяв за левый и правый аргументы, соответственно `$a` и `$b`, получаем результаты, перечисленные в табл. 1.5.

Таблица 1.5. Операторы сравнения

Сравнение	Числа	Строки	Возвращаемое значение
Равно	==	eq	Истина, если $a равно $b
Не равно	!=	ne	Истина, если $a не равно $b
Меньше	<	lt	Истина, если $a меньше $b
Больше	>	gt	Истина, если $a больше $b
Меньше или равно	<=	le	Истина, если $a не больше $b
Больше или равно	>=	ge	Истина, если $a не меньше $b
Сравнение	<=>	cmp	0, если $a и $b равны, 1, если $a больше, -1, если $b больше

Последние два оператора (`<=>` и `cmp`) совершенно избыточны. Однако они крайне полезны в подпрограммах сортировки `sort` (см. главу 27).[1]

[1] Кое-кто воспринимает такую избыточность как порочную, поскольку из-за нее язык перестает быть «минималистским», или ортогональным. Но Perl и не является ортогональным языком, это диагональный язык. Мы подразумеваем под этим, что Perl не вынуждает программиста всегда перемещаться под прямым углом. Иногда хочется попасть в нужное место по гипотенузе треугольника. TMTOWTDI предполагает возможность срезать углы. Возможность срезать углы работает на эффективность.

Некоторые операторы проверки файлов

Операторы этой группы позволяют проверить атрибуты файлов, чтобы работать с этими файлами не вслепую. Основным атрибутом файла является, конечно, факт его существования. Например, весьма уместно поинтересоваться, не существует ли уже у вас файл с почтовыми псевдонимами, прежде чем открывать его как новый файл, удаляя при этом все имеющиеся в нем данные. В табл. 1.6 перечислено несколько операторов проверки файлов:

Таблица 1.6. Операторы проверки файлов

Пример	Название	Результат
`-e $a`	Существует (exists)	Истина, если файл с именем $a существует
`-r $a`	Доступен для чтения (readable)	Истина, если файл с именем $a доступен для чтения
`-w $a`	Доступен для записи (writable)	Истина, если файл с именем $a доступен для записи
`-d $a`	Каталог (directory)	Истина, если файл с именем $a является каталогом
`-f $a`	Файл (file)	Истина, если файл с именем $a является обычным файлом
`-T $a`	Текстовый файл (text file)	Истина, если файл с именем $a является текстовым файлом

Можно так использовать эти операторы:

```
-e "/usr/bin/perl" or warn "Perl неправильно установлен\n";
-f "/vmlinuz" and say "Вижу, что вы друг Линуса";
```

Обратите внимание, что обычный (regular) файл – не то же самое, что текстовый файл (text file). Двоичные файлы типа */vmlinuz* являются обычными, но не текстовыми. Текстовые файлы представляют собой противоположность двоичным файлам, тогда как обычные файлы – противоположность «иррегулярным» файлам типа каталогов или устройств.

Существует много операторов проверки файлов помимо перечисленных нами. Большинство таких операторов являются унарными логическими операторами, т.е. принимают только один операнд (скаляр, значением которого является имя файла или его дескриптор) и возвращают значение `true` или `false`. Некоторые из них возвращают более замысловатые значения, например, размер или время, относящееся к файлу; их, когда они вам понадобятся, вы сможете найти в разделе «Именованные унарные операторы и операторы проверки файлов» в главе 3.

Управляющие конструкции

До настоящего момента все рассмотренные нами примеры, за исключением одного большого примера, были совершенно прямолинейными: мы выполняли все команды по порядку. Мы привели несколько примеров использования операторов короткого пути для выполнения (или невыполнения) отдельной команды. Хотя можно писать очень полезные линейные программы (к ним относятся многие сценарии CGI), при наличии условных выражений и циклов можно создавать значительно более мощные приложения. В совокупности эти механизмы называются

управляющими конструкциями. Поэтому можно рассматривать Perl и как управляющий язык.

Но чтобы управлять, необходимо иметь возможность принимать решения, а чтобы принимать решения, нужно знать, в чем разница между истиной и ложью.

Что есть истина?

Мы обсуждали термин «истинность» (truth)[1] и упоминали, что некоторые операторы возвращают значения true или false. Прежде чем двигаться дальше, необходимо точно договориться, что под ними подразумевается. В Perl значение истинности рассматривается несколько иначе, чем в других языках программирования, но, поработав с ним некоторое время, читатель найдет это очень разумным. (На самом деле, мы надеемся, что разумным это покажется вам после чтения нижеизложенного.)

В сущности, Perl считает, что истина самоочевидна. Это поверхностный способ сказать, что можно оценить истинность почти чего угодно. В Perl используется практическое определение истинности, зависящее от типа того, что подвергается оценке. Оказывается при этом, что типов истинности существует значительно больше, чем типов неистинности.

Истинность в Perl всегда вычисляется в скалярном контексте. Какого-либо другого приведения типов не производится. Вот правила для различных типов значений, которые может содержать скаляр:

1. Любая строка, кроме "" и "0", является истиной.
2. Любое число, кроме 0, является истиной.
3. Любая ссылка является истиной.
4. Любая неопределенная величина является ложью.

Фактически последние два правила следуют из двух первых. Любая ссылка (правило 3) указывает на нечто, имеющее адрес, и имеет значением число или строку, содержащую этот адрес, который не может быть равен 0, поскольку всегда определен. А всякая неопределенная величина (правило 4) всегда имеет значением 0 или пустую строку.

А в некотором смысле и правило 2 можно вывести из правила 1, если представить, что все величины являются строками. Опять же, для вычисления истинности никакого приведения строк не производится, но если бы приведение строк производилось, число 0 превратилось бы в строку "0" и оказалось бы ложью. Любое другое число превратилось бы в строку, отличную от "0", а потому оказалось бы истиной. Рассмотрим несколько примеров, которые позволят лучше это понять:

```
0          # становится строкой "0", поэтому false
1          # становится строкой "1", поэтому true.
10 - 10    # 10 минус 10 равно 0, преобразуется в строку "0", поэтому false.
0.00       # равно 0, преобразуется в строку "0", поэтому false.
"0"        # является строкой "0", поэтому false
""         # является пустой строкой, поэтому false.
"0.00"     # является строкой "0.00", ни "", ни "0", поэтому true!
"0.00" + 0 # становится числом 0 (приведение из-за +), поэтому false.
```

[1] Строго говоря, это не истина.

```
\$a        # является ссылкой на $a, поэтому true, даже если $a есть false
undef()    # функция, возвращающая неопределенное значение, поэтому false.
```

Поскольку ранее было что-то невнятно сказано о вычислении истинности в скалярном контексте, у читателя может возникнуть вопрос о том, как оценивается истинность списка. Отвечаем: ни одна операция Perl не возвращает список в скалярном контексте. Обнаружив скалярный контекст, все они возвращают скалярное значение, для которого уже применяются правила истинности. Поэтому здесь не возникает проблемы, ведь можно определить значение, возвращаемое любым таким оператором в скалярном контексте. Оказывается, как массивы, так и хеши возвращают скалярные значения, являющиеся истиной, если в массиве или хеше есть хоть один элемент. Подробнее об этом далее.

Операторы if и unless

Мы видели выше, что логический оператор может действовать в качестве условного. Несколько более сложной формой логического оператора является оператор if. Оператор if вычисляет истинность условия (т.е. логическое выражение) и выполняет блок кода, если условие истинно:

```
if ($debug_level > 0) {
    # Что-то не так. Сообщить пользователю.
    say "Отладка: Тревога, Уилл Робинсон, тревога!";
    say "Отладка: Ответ был '54', а ожидалось '42'.";
}
```

Блоком называется один или несколько операторов, заключенных в пару фигурных скобок. Поскольку оператор if выполняет блок, фигурные скобки требуются в нем по определению. Те, кто знаком с языком типа C, заметят отличие. Скобки необязательны в C, если команда в блоке всего одна, но в Perl скобки нужны всегда.

Бывает, что недостаточно выполнить блок команд, если условие истинно. Может потребоваться выполнить другой блок, когда условие *ложно*. Конечно, можно использовать два оператора if таких, что условие второго будет отрицанием условия первого, но Perl предоставляет более элегантное решение. За блоком if может следовать необязательное второе условие, называемое else, которое выполняется, только если условие if ложно. (Ветеранов программирования это не удивит.)

Иногда возможных вариантов оказывается больше двух. В таких случаях требуется добавить условие истинности elsif для каждого возможного варианта. (Ветеранов программирования может весьма удивить орфография ключевого слова elsif, но за нее мы не собираемся просить прощения. Извините.)

```
if ($city eq "New York") {
    say "Нью-Йорк находится к северо-востоку от Вашингтона, округ Колумбия.";
}
elsif ($city eq "Chicago") {
    say "Чикаго находится к северо-западу от Вашингтона, округ Колумбия.";
}
elsif ($city eq "Miami") {
    say "Майами находится к югу от Вашингтона. И там гораздо теплее!";
}
else {
    say "Извините, но я не знаю, где находится $city.";
}
```

Предложения if и elsif вычисляются по очереди, пока одно из них не окажется истинным или не будет достигнуто условие else. Когда одно из условий оказывается истинным, выполняется его блок, а все остальные ветви пропускаются. Иногда требуется, чтобы никакие действия не выполнялись, если условие истинно – только если оно ложно. Применение пустого if с else выглядит неряшливо, а if с отрицанием затруднять чтения кода; в обычном языке фраза «если не это является истиной, сделать то-то и то-то» звучит несколько странно. В таких случаях следует предпочесть оператор unless[1]:

```
unless ($destination eq $home) {
    say "Я еду не домой.";
}
```

Оператор elsunless отсутствует. Обычно это засчитывается как функция языка.

Операторы given и when

Чтобы обеспечить возможность сравнения единственного значения с множеством альтернатив, в последних версиях Perl появились инструкции, которые в других языках программирования обычно называются switch и case. Однако нам нравится, когда Perl действует подобно естественному языку, так что мы назвали их given и when. (Так как вы уже добавили в начало программы директиву use v5.14, эта функциональность, появившаяся в версии 5.10, должна быть доступна.)

```
#!/usr/bin/perl
use v5.14;

print "Ваш любимый цвет? ",
chomp(my $answer = <STDIN>);

given ($answer) {
    when ("фиолетовый")   { say "Мой тоже." }
    when ("зеленый")      { say "Вперед!" }
    when ("желтый")       { say "Внимание!" }
    when ("красный")      { say "Стой!" }

    when ("синий")        { say "Продолжайте." }
    when (/\w+, нет \w+/) { die "ХРЯСЬ!" }

    when (42)             { say "Неправильный ответ

    when (['серый','оранжевый , коричневый'  черный', белый']) {
        say "Я думаю, что $answer тоже неплохо.
    }

    default {
        say "Вы уверены, что $answer - это цвет?";
    }
}
```

Сначала инструкция given вычисляет значение своего выражения и объявляет его темой беседы, благодаря чему последующие инструкции when знают, какое значение сверять. Затем выполняется сопоставление аргументов каждой инструкции

---

1 «Если не», англ. – *Прим. ред.*

when, чтобы отыскать первую инструкцию when, значением которой соответствует теме. Сопоставление аргументов инструкций when выполняется по порядку, сверху вниз, и прекращается, как только будет найдено первое соответствие – сопоставление с остальными инструкциями when не производится, а управление передается сразу за пределы конструкции given.

Форма аргумента каждой инструкции when ("красный" или 42 или /\w+, нет \w+/) определяет тип выполняемого сопоставления. Проще говоря, строки сопоставляются как строки, числа – как числа, а шаблоны – ну... как шаблоны. При сопоставлении со списками соответствие считается состоявшимся, если совпадает хотя бы одно значение. Инструкции when внутренне выполняют операцию, получившую название «интеллектуальное сопоставление» (smartmatching)[1]. Она выполняет сопоставление ожидаемым способом, за исключением случаев, когда решает, что нужен иной способ. Подробнее об этом рассказывается в разделе «Оператор интеллектуального сопоставления» в главе 3.

Итеративные (циклические) конструкции

Эти операторы позволяют программе Perl повторять выполнение одного и того же кода, за что и получили название *итеративные* (*циклические*) конструкции. Существует несколько типов таких конструкций, отличающихся в основном способом определения момента, когда следует выйти из цикла и заняться другими делами.

Циклы с условием

Операторы while и until ведут себя аналогично операторам if и unless, но позволяют выполнять блок многократно, пока условие соблюдается. Условие всегда проверяется перед началом очередной итерации. Если условие соблюдено (принимает значение true для while или false для until), выполняется блок операторов цикла.

```
print "Сколько билетов уже продано?
my $before = <STDIN>;

my $sold = $before;
while ($sold < 10000) {
    my $available = 10000 - $sold;
    print "$available билетов в наличии. Сколько нужно вам:
    my $purchase = <STDIN>;
    if ($purchase > $available) {
        say "Слишком много! Попробуйте еще раз.";
        $purchase = 0;
    }
    $sold += $purchase;
}

say "Билеты закончились, приходите в ругой раз.";
```

Обратите внимание, что если исходное условие не выполнено, вход в цикл вообще не производится. Например, если уже продано 10 000 билетов, программа сразу же сообщит, что все билеты проданы.

---

[1] Иногда оператор интеллектуального сопоставления, о котором пойдет речь в главе 3, называют еще оператором нечеткого сопоставления. – *Прим. ред.*

В строке 8 нашего примера вычисления среднего осуществляется чтение:

```
while (my $line = <GRADES>) {
```

В результате очередная строка присваивается переменной `$line` и, как объяснялось выше, возвращается значение `$line`, поэтому в условии оператора `while` может быть определена истинность `$line`. Читателю может показаться, что Perl получит ошибочное ложное значение при вводе пустой строки и досрочно выйдет из цикла. На самом деле этого не произойдет. Причина будет понятна, если поразмыслить обо всем сказанном выше. Оператор чтения строки оставляет в конце строки символ перевода строки, поэтому пустая строка имеет значение `"\n"`. А как известно, `"\n"` не является каким-либо из канонических ложных значений. Поэтому условие истинно и цикл продолжается даже при вводе пустых строк.

С другой стороны, когда в конце концов будет достигнут конец файла, оператор чтения строки вернет неопределенное значение, которое всегда ложно. В результате выполнение цикла прекращается там, где это нам и нужно. В Perl не требуется явно проверять значение функции `eof`, поскольку операторы чтения ввода разработаны так, чтобы гладко работать в контексте условных операторов.

На практике почти все спроектировано так, чтобы гладко работать в условном (логическом) контексте. Если обратиться к массиву в скалярном контексте, возвращается размер массива. Поэтому часто встречается такая обработка аргументов командной строки:

```
while (@ARGV) {
    process(shift @ARGV),
}
```

Оператор `shift` при каждой итерации цикла удаляет один элемент из списка аргументов (и возвращает его). Цикл автоматически завершается, когда исчерпывается массив `@ARGV`, т.е. его длина становится равной `0`. А `0` в Perl является ложью. В некотором смысле сам массив становится ложью.[1]

Трехчастный цикл

Другой итеративный оператор – трехчастный цикл, также известный, как цикл `for` в стиле C. Трехчастный цикл выполняется точно так же, как цикл `while`, но выглядит совсем иначе из-за того, что в определение цикла было добавлено две дополнительные инструкции. (Однако программистам на C он покажется очень знакомым.)

```
print " Сколько билетов уже продано?
my $before = <STDIN>,

for (my $sold = $before; $sold < 10000; $sold += my $purchase) {
    my $available = 10000 - $sold;
    print "$available билетов в наличии. Сколько нужно вам  ";
```

---

[1] Это стиль мышления программистов на Perl. Поэтому не нужно сравнивать 0 и 0, чтобы проверить, не является ли значением этого выражения «ложь». Несмотря на то что в других языках это требуется, не вступайте на путь выполнения явных сравнений типа `while (@ARGV != 0)`. Это неэффективно как для вас, так и для компьютера. Да и для каждого, кому придется сопровождать ваш код.

```
    $purchase = <STDIN>;
    if ($purchase > $available) {
        say "Слишком много! Попробуйте еще раз.";
        $purchase = 0;
    }
}

say " Билеты закончились, приходите в ругой раз.";
```

Трехчастный цикл принимает три выражения (откуда и взялось такое название), заключенных в круглые скобки и разделенных символом точка с запятой. Первое выражение выполняет начальную установку переменной цикла. Второе – условие для проверки переменной цикла – действует так же, как в цикле while. И третье выражение изменяет состояние переменной цикла – это выражение выполняется в конце каждой итерации, подобно явной инструкции в конце цикла while в примере выше.

В начале цикла устанавливается исходное значение и проверяется истинность условия. Если условие истинно, выполняется блок. Когда блок кончается, выполняется модифицирующее выражение, снова проверяется истинность условия, и, если условие выполнено, блок выполняется снова с измененным значением управляющей переменной. Пока условие сохраняет истинность, будут выполняться и блок, и модифицирующее выражение. (Обратите внимание, что только среднее выражение вычисляется с целью получения его значения. Первое и третье выражения вычисляются только ради получения побочного эффекта, а их значения отбрасываются!)

Любое из трех выражений можно опустить, но две точки с запятой должны присутствовать всегда. Если опустить выражение в середине, получится бесконечный цикл, т.е. бесконечный цикл можно записать так:

```
for (;;) {
    say "Вынеси мусор!"
    sleep(5);
}
```

Цикл foreach

Последним из итеративных операторов Perl является цикл *foreach*,[1] применяемый для выполнения одного и того же кода для каждого скаляра из некоторого набора, которым может, например, быть массив:

```
for my $user (@users) {
    if (-f "$home{$user}/.nexrc") {
    say "$user крут . он использует vi с поддержкой perl!"
    }
}
```

[1] Исторически он записывался с помощью ключевого слова foreach, откуда и взялось название цикла. В настоящее время предпочтение отдается ключевому слову for, поскольку такая инструкция читается более естественно, когда включается объявление my, и не дает спутать синтаксис этого цикла с трехчастным циклом. Поэтому многие из нас больше не используют ключевое слово foreach, хотя вы можете продолжать пользоваться им, если оно вам нравится.

В отличие от операторов if и while, которые обеспечивают скалярный контекст в условном выражении, цикл *foreach* обеспечивает списочный контекст для выражения в круглых скобках. Поэтому в результате вычисления выражения получается список (не скаляр, даже если в списке лишь один скаляр). После этого переменная цикла поочередно получает имена всех элементов списка, и блок кода выполняется по одному разу для каждого элемента списка. Обратите внимание, что переменная цикла ссылается на сам элемент, а не на его копию. В результате изменение переменной цикла приводит к изменению исходного массива.

В типичной программе на Perl можно обнаружить значительно больше циклов *foreach*, чем трехчастных циклов for, поскольку в Perl очень легко создавать такие списки, которые требуются циклу *foreach* для выполнения итераций. (Отчасти именно поэтому мы используем ключевое слово for – мы ленивы и считаем, что часто используемые слова должны быть как можно короче.) Вы часто будете встречать идиому перебора отсортированных ключей хеша в цикле:

```
for my $key (sort keys %hash) {
```

На практике именно это делает строка 13 нашего Примера вычисления среднего, благодаря которой список студентов выводится в алфавитном порядке.

Выход из цикла: next и last

Операторы next и last позволяют изменять порядок выполнения команд в цикле. Не так уж редки особые ситуации, которые нужно пропускать или при возникновении которых нужно завершить цикл. Например, при работе с учетными записями в UNIX может потребоваться пропустить системные учетные записи, такие как *root* или *lp*. Оператор next позволяет перейти в конец текущей итерации цикла и начать новую итерацию. Оператор last позволяет перескочить конечную границу блока, как если бы условие выполнения цикла вдруг оказалось ложным. Это может быть полезно, если требуется, к примеру, отыскать определенную учетную запись и, найдя ее, выйти из цикла.

```
for my $user (@users) {
    if ($user eq "root" || $user eq "lp") {
        next;
    }
    if ($user eq "special") {
        print "Найдена учетная запись special.\n";
        # какая-то обработка
        last;
    }
}
```

Существует возможность прерывать выполнение вложенных циклов с помощью меток циклов, позволяющих поименно указывать циклы для выхода. При использовании модификаторов операторов (другого вида условных операторов, о котором речь пойдет ниже) это позволяет создавать исключительно понятные конструкции выхода из циклов (если вы согласны, что английский язык – понятный):

```
LINE: while (my $line = <EMAIL>) {
    next LINE if $line =~ "\n"; # пропустить пустые строки
    last LINE if $line =~ /^>/; # остановиться на первой строке в кавычках
```

```
    # здесь могла бы быть ваша реклама
}
```

Читатель может сказать: «Погодите, что это еще за нелепые значки =~ между покосившимися зубочистками? Это не очень-то похоже на английский язык.» И будет прав. Это шаблон для поиска, содержащий регулярное выражение (хотя и довольно простое). О регулярных выражениях будет рассказано в следующем разделе. Perl является лучшим в мире языком обработки текста, а регулярные выражения – основной механизм обработки текста в Perl.

Регулярные выражения

Регулярные выражения (*regular expressions*) (известные также как regexes, regexps или RE) используются многими программами поиска, такими как *grep* и *findstr*, пакетными редакторами текста вроде *sed* и *awk*, а также редакторами – например, *vi* и *emacs*. Регулярное выражение – это способ описать множество строк, не перечисляя все строки, входящие в это множество.[1] Регулярные выражения входят во многие другие языки программирования (некоторые из них даже рекламируются как поддерживающие «регулярные выражения Perl5»!), но ни в один из этих языков регулярные выражения не интегрированы так, как в Perl. Регулярные выражения используются в Perl несколькими способами. Во-первых, и это главное, они применяются в условных операторах, чтобы определить, соответствует ли строка некоторому шаблону, поскольку в логическом контексте они возвращают «истину» или «ложь». Поэтому, увидев в условном операторе нечто вроде /foo/, знайте, что перед вами обычный оператор *сопоставления с шаблоном* (pattern-matching operator):

```
if (/Windows 7/) { print "Пора делать апгрейд?\n" }
```

Во-вторых, найдя в строке соответствие шаблону, можно заменить его чем-то другим. Поэтому, увидев нечто типа s/foo/bar/, знайте, что это задание Perl подставить при возможности «bar» вместо «foo». Мы называем это оператором *подстановки* (substitution). Он также возвращает true или false в зависимости от успешности операции, но обычно выполняется ради своего побочного эффекта:

```
s/IBM/lenovo/;
```

Наконец, с помощью шаблонов можно определять не только факт существования, но и факт *отсутствия*. Так, оператор split использует регулярное выражение, чтобы определять, где данных нет. Это значит, что регулярное выражение задает *разделители* (*separators*), разграничивающие поля данных. В нашем Примере вычисления среднего есть пара простейших образцов. В строках 9 и 16 производится расщепление строк по пробелам и возвращается список слов. Но расщепление возможно и по любому другому разделителю, заданному регулярным выражением:

```
my ($good, $bad, $ugly) = split(/,/, "vi,emacs,teco");
```

---

[1] Хорошим источником информации по концепциям регулярных выражений служит книга Джеффри Фридла (Jeffrey Friedl) «Mastering Regular Expressions» («Регулярные выражения», 3-е издание, Символ-Плюс, 2008).

(Существуют различные модификаторы, которые можно применять в каждом из этих случаев для осуществления экзотических действий. К примеру, можно предписать Perl не обращать внимания на регистр при поиске символов алфавита, но эти кровавые подробности мы будем освещать во второй части книги, когда доберемся до кровавых подробностей.)

В простейшем случае регулярные выражения используются для поиска литерального выражения. В приведенном выше примере расщепления осуществлялся поиск одиночных запятых. Но если надо отыскать несколько символов подряд, то и в строке они должны идти один за другим. То есть шаблон, как можно догадаться, ищет подстроку. Допустим, что требуется показать все строки файла HTML, содержащие ссылки HTTP (но не ссылки FTP). Представим себе, что мы впервые работаем с файлом HTML и слегка наивны. Мы знаем, что в этих ссылках всегда где-то присутствует "`http:`". Можно организовать такой цикл по нашему файлу:

```
while (my $line = <FILE>) {
    if ($line =~ /http:/) {
        print $line;
    }
}
```

Здесь =~ (оператор привязки шаблона) указывает Perl на необходимость поиска соответствий регулярному выражению "`http:`" в значении переменной `$line`. Если соответствие найдется, оператор вернет значение «истина» и будет выполнен блок (оператор `print`).[1]

Между прочим, если не использовать оператор привязки =~, то Perl будет осуществлять поиск в строке, установленной по умолчанию, а не в `$line`. Это все равно, что сказать: «Ой! Помогите найти мою контактную линзу!» И люди станут автоматически искать ее около вас, хотя вы не говорили им, где нужно искать. Perl тоже знает, что есть место поиска по умолчанию, если не указано, где надо искать. Эта строка по умолчанию является на самом деле особой скалярной переменной с необычным именем $_. На практике это место по умолчанию используется не только для поиска по шаблону; многие операторы Perl используют по умолчанию переменную $_, поэтому опытный программист на Perl, вероятно, записал бы последний пример так:

```
while (<FILE>) {
    print if /http:/
}
```

(Хм-м, похоже, что сюда проник еще один из этих модификаторов операторов. Хитрые маленькие бестии.)

Все это достаточно удобно, но что если нужно найти все типы ссылок, а не только ссылки HTTP? Можно дать список типов ссылок, например, "`http:`", "`ftp:`", "`mailto`" и т.д. Но этот список может оказаться длинным, да и как быть, если появится новый тип ссылок?

```
while (<FILE>) {
    print if /http:/;
    print if /ftp:/;
```

[1] Это очень похоже на то, что делает команда UNIX `grep 'http:' file`.

```
        print if /mailto:/;
        # Что еще?
    }
```

Поскольку регулярное выражение описывает множество строк, можно просто описать, что мы ищем: несколько символов алфавита, за которыми следует двоеточие. На языке регулярных выражений (регулярском?) это будет выглядеть как /[a-zA-Z]+:/, где квадратные скобки определяют *класс символов* (character class). Группы a-z и A-Z представляют все символы английского алфавита (дефис обозначает диапазон всех символов с начального по конечный включительно). А знак + является специальным символом, означающим «не менее одного из того, что мне предшествовало». Это то, что мы называем *квантификатором* (quantifier), представляющим собой штуковину, позволяющую сообщить, сколько раз нечто может повториться. (Символ косой черты не является частью регулярного выражения, это часть оператора сопоставления с шаблоном. Пара таких косых черт действует подобно кавычкам, в которые заключается регулярное выражение.)

Поскольку некоторые классы, например, символы алфавита, используются очень часто, Perl определяет для них сокращения, перечисленные в табл. 1.7.

Таблица 1.7. Сокращения, представляющие алфавитные символы

Название	Определение в ASCII	Определение в Юникоде	Код
Пробельный символ	[\t\n\r\f]	\p{Whitespace}	\s
Символ слова	[a-zA-Z_0-9]	[\p{Alphabetic}\p{Digit}\p{Mark}\p{Pc}]	\w
Цифра	[0-9]	\p{Digit}	\d

Заметьте, что эти коды соответствуют одиночным символам. Метасимвол \w ищет соответствие любому одиночному символу слова, а не целому слову. (Квантификатор + еще не забыт? Можно задать \w+ для поиска целого слова.) В Perl можно задать отрицание этих классов, используя символы верхнего регистра: например, \D означает символ, не являющийся цифрой.

Следует заметить, что \w не всегда эквивалентно [a-zA-Z_0-9] (а \d не всегда эквивалентно [0-9]). В некоторых наборах параметров локализации (locales) определяются дополнительные символы, не входящие в ASCII, и метасимвол \w в курсе таких дополнений. Новым версиям Perl (выше 5.8.1) «знакомы» также свойства букв и цифр Юникода, и символы Юникода с этими свойствами обрабатываются надлежащим образом. (Perl даже идеограммы и объединяющие символы рассматривает как символы \w.)

Есть еще один совершенно особенный класс символов, записываемый как точка (.), которому соответствует вообще любой символ.[1] Например, /a./ соответствует любой строке, содержащей a, если только это не последний символ в строке. Таким образом, это выражение соответствует at или am, или даже a!, но не a, поскольку после a нет ничего, чему могла бы соответствовать точка. Поскольку поиск по шаблону выполняется в любом месте строки, ему соответствуют также oasis и camel, но не sheba. Он соответствует первому символу a в слове caravan. Он

[1] За исключением того, что обычно символ перевода строки ему не соответствует. Можно вспомнить, что (.) обычно не соответствует переводу строки и в *grep*(1).

мог бы соответствовать и второму символу a, но поиск (ведущийся слева направо) останавливается после нахождения первого соответствия.

Квантификаторы

Символы и классы символов, о которых мы говорили, ищут соответствие одиночным символам. Мы уже говорили, что можно искать соответствие нескольким символам «слова» посредством \w+. Одним из квантификаторов является +, но есть и другие. Любой квантификатор следует за элементом, количество вхождений которого определяет.

В самом общем виде квантификатор задает минимальное и максимальное допустимое число соответствий элемента. Эти два числа заключаются в скобки и разделяются запятой. Например, если пытаться найти номера телефонов в Северной Америке, то последовательность \d{7,11} соответствует не менее чем семи цифрам, но не более чем одиннадцати. Если в скобки помещено одно число, оно задает одновременно минимум и максимум, т.е. точное число соответствий элемента. (Все элементы без указания количества имеют неявный квантификатор {1}.)

Если указать минимум и запятую, но опустить максимум, то максимум принимается равным бесконечности. Иными словами, соответствие должно быть найдено хотя бы минимальное число раз, плюс еще сколько угодно. Например, \d{7} будет соответствовать только первым семи цифрам (к примеру, местному номеру в Северной Америке или первым семи цифрам более длинного номера), в то время как \d{7,} будет соответствовать любому номеру телефона, даже международному (если только он не окажется короче семи цифр). Нет специального способа задания «не более чем» определенного числа раз. Нужно просто сказать, например, {0,5}, чтобы найти не более пяти произвольных символов.

Некоторые сочетания минимума и максимума встречаются так часто, что в Perl для них имеются особые квантификаторы. Мы уже видели +, означающий то же, что {1,}, или «хотя бы один из предшествующих элементов». Имеется также *, что равносильно {0,}, или «ноль или более предшествующих элементов», а также ?, что равносильно {0,1}, или «ноль или один предшествующий элемент» (иначе говоря, предшествующий элемент является необязательным).

Применяя квантификаторы, следует кое в чем проявлять осторожность. Во-первых, квантификаторы Perl по умолчанию являются *жадными*. Это означает, что они ищут самое длинное соответствие шаблону из всех возможных. Например, если искать /\d+/ в строке 1234567890, то будет найдена вся строка. За этим нужно следить, особенно если используется символ , соответствующий любому символу. Часто встречаются ситуации, когда в строке типа

```
larry:JYHtPh0./NJTU:100:10:Larry Wall:/home/larry:/bin/tcsh
```

выполняется поиск larry: с помощью / +:/. Однако, поскольку квантификатор + является жадным, соответствие шаблону включит все до конца подстроки /home/larry:, поскольку отыскивается самое длинное соответствие перед последним двоеточием, включая другие двоеточия. Иногда этого удается избежать посредством отрицания класса, т.е. с помощью выражения /[^:]+:/, которое требует найти один или несколько символов (как можно больше), не являющихся двоеточием, вплоть до первого двоеточия. Эта маленькая «крышечка» в регулярном выраже-

нии служит для отрицания класса символов.[1] Другой момент, который надо иметь в виду, – это стремление регулярных выражений находить как можно более *раннее* соответствие. Оно даже берет верх над жадностью. Поскольку просмотр строки происходит слева направо, это означает, что соответствие шаблону будет найдено как можно левее, даже если в другом месте есть более длинное соответствие. (Регулярные выражения жадны, но вознаграждение предпочитают получать как можно быстрее.) Предположим, например, что используется команда подстановки (s///) в строке по умолчанию, т.е. в отношении переменной $_, и требуется удалить последовательность символов x из середины строки. Если сказать:

```
$_ = "fred xxxxxxx barney";
s/x*//;
```

эффекта не будет никакого! Это связано с тем, что x* (означающее ноль или более символов x) соответствует «ничему» в начале строки, поскольку пустая строка имеет длину ноль символов, а ясно, как божий день, что перед символом f в слове fred сидит пустая строка.[2] И еще об одной вещи нужно знать. По умолчанию квантификаторы применяются к единственному предшествующему символу, поэтому /bam{2}/ найдет bamm, но не bambam. Чтобы применить квантификатор к нескольким символам, поставьте скобки. Для поиска bambam используйте шаблон /(bam){2}/.

Минимальное соответствие

При работе с доисторическими версиями Perl, когда не нужен был жадный поиск, приходилось использовать отрицание класса. (На самом деле все равно получался жадный поиск, но несколько более сдержанный.)

В новых версиях Perl можно принудительно задать нежадный, минимальный поиск, поместив после любого квантификатора вопросительный знак. Теперь тот же наш поиск имени пользователя должен выглядеть как /.*?:/. Этот .*? будет теперь стараться найти не как можно больше символов, а как можно меньше, поэтому остановка произойдет после первого двоеточия, а не последнего.

Как поймать на слове

При попытке найти соответствие шаблону проверка осуществляется с каждого места, пока не будет найдено соответствие. *Якорь* (anchor) позволяет ограничить область поиска совпадений с шаблоном. В сущности, якорь есть нечто, соответствующее «ничему», но это «ничто» особое, оно зависит от окружения. Можно назвать это «ничто» также правилом, ограничением, утверждением. Как ни называй, якорь пытается найти нечто нулевой длины, успешно или не очень. (Отсутствие соответствия якорю просто означает, что соответствие шаблону нельзя найти этим конкретным способом. Если можно попробовать еще какие-то способы, то будут продолжены попытки найти шаблон этими способами.)

Специальный символ \b соответствует границе слова, которая определяется как «ничто» между символом слова (\w) и символом, не принадлежащим слову (\W),

---

1 Мы приносим извинения, но это обозначение выбрано не нами, и не стоит винить нас. Отрицание классов символов традиционно записывается именно так в культуре UNIX.

2 Не грустите. Даже авторы иногда спотыкаются о подобные вещи.

в любом порядке. (Символы, «предшествующие» началу строки или «следующие» после конца строки, не считаются символами слова.) Например:

```
/\bFred\b/
```

будет соответствовать слову Fred как в строке "The Great Fred", так и в строке "Fred the Great", но не в строке "Frederick the Great", потому что за d в слове Frederick не следует «символ, не принадлежащий слову».

В аналогичное русло укладываются якоря, соответствующие началу или концу строки. Символ (^), будучи первым символом строки, соответствует «ничему» в начале строки. Поэтому шаблон /^Fred/ будет соответствовать строке "Fred в Frederick the Great", но не в строке "The Great Fred", тогда как /Fred^/ не найдется ни в одной из этих строк. (На самом деле, в последнем выражении вообще мало смысла.) Знак доллара ($) работает так же, как ^, но соответствует «ничему» не в начале, а в конце строки.[1] Теперь, вероятно, читатель уже сможет определить, что фраза

```
last LINE if $line =~ /^>/;
```

означает «перейти к последней итерации цикла LINE, если данная строка начинается с символа >».

Выше мы сказали, что последовательность \d{7,11} будет соответствовать числу длиной от семи до одиннадцати цифр. Будучи абсолютно верным, это утверждение вводит в заблуждение: задание этой последовательности в действительном операторе поиска по шаблону, /\d{7,11}/, не исключает того, что после одиннадцати соответствующих шаблону цифр не окажется других, ему не соответствующих! Для получения желаемого результата часто требуется добавлять якори в шаблоны с одного или обоих концов.

Обратные ссылки

Ранее говорилось, что объединять элементы, на которые действует квантификатор, можно с помощью скобок, но можно использовать скобки и для запоминания фрагментов найденного. Если заключить часть регулярного выражения в пару скобок, символы, соответствующие этой части в найденном, будут сохранены и смогут использоваться в дальнейшем. При этом критерии поиска не изменяются, поэтому /\d+/ и /(\d+)/ будут искать самую длинную последовательность цифр, но в последнем случае найденная последовательность будет сохранена в специальной переменной, к которой позже можно обратиться.

Способ обращения к найденной части строки зависит от того, откуда требуется выполнить это обращение. Внутри того же самого регулярного выражения это делается с помощью обратной косой черты, за которой следует целое число. Целое число, соответствующее данной паре скобок, определяется путем подсчета числа левых скобок с начала шаблона, начиная с единицы. Например, для поиска чего-то похожего на тег HTML, типа <B>Bold</B>, можно использовать шаблон /<(.*?)>.*?<\/\1>/. Он требует, чтобы обе его части в точности соответствовали одинаковой строке, как "B" в данном примере.

---

[1] Это некоторое упрощение, поскольку предполагается, что строка не содержит символ перевода строки; ^ и $ являются, в сущности, якорями для начала и конца строк в тексте (lines), а не последовательностей символов (strings). Мы постараемся разобраться со всем этим в главе 5 (насколько с этим вообще можно разобраться).

Вне самого регулярного выражения, например, в части замены для оператора подстановки, используется сочетание $ и целого числа, т.е. обычная скалярная переменная, именем которой является число. Поэтому, чтобы переставить местами первые два слова в строке, можно использовать оператор:

```
s/(\S+)\s+(\S+)/$2 $1/
```

В правой части подстановки (между вторым и третьим символами косой черты) находится просто необычная разновидность строки в двойных кавычках, поэтому и можно производить в ней интерполяцию переменных, включая переменные с найденным текстом. Это мощный механизм: интерполяция (в контролируемых условиях) – одна из причин, почему Perl является хорошим языком обработки текста. Другой причиной, само собой, является поиск по шаблону. Регулярные выражения предназначены для того, чтобы разобрать на кусочки, а интерполяция – чтобы собрать все вместе снова. Возможно, не все потеряно для Шалтая-Болтая.

Если использование нумерованных ссылок покажется вам утомительным, в версии Perl v5.10 и выше поддерживаются также именованные ссылки. Это те же самые подстановки, но на этот раз с использованием именованных групп:

```
s/(?<alpha>\S+)\s+(?<beta>\S+)/$+{beta} $+{alpha}/
```

Возможно, их применение увеличит время набора текста программы, но как только размеры и сложность ваших шаблонов подрастет, вы будете рады возможности именовать группы осмысленными словами вместо простых чисел.

Таблица 1.8. Ссылки в регулярных выражениях

Где	Нумерованная группа	Именованная группа
Объявление	(..)	(?<*ИМЯ*>
Внутри того же регулярного выражения	\1	\k<*ИМЯ*>
В программном коде на Perl	$1	$+{*ИМЯ*}

Обработка списков

Где-то в начале этой главы мы говорили, что в Perl есть два основных контекста: скалярный контекст (для работы с отдельными объектами) и списочный контекст (для работы с наборами объектов). Многие из традиционных операторов, встречавшихся в тексте до сих пор, были по своему действию чисто скалярными. Они всегда принимают одиночные аргументы (или пары одиночных аргументов в бинарных операторах) и всегда производят одиночный результат, даже в списочном контексте. Поэтому, если есть запись:

```
@array = (1 + 2, 3 - 4, 5 * 6, 7 / 8);
```

то список в правой части содержит ровно четыре значения, поскольку результатом обычных математических операторов всегда являются скалярные величины, даже в списочном контексте, возникающем при присваивании массиву.

Однако другие операторы Perl могут порождать как скалярные, так и списочные значения, в зависимости от контекста. Они просто «знают», что от них ожидается – скаляр или список. Но как об этом узнает программист? Оказывается, это нетрудно сделать, если усвоить некоторые ключевые понятия.

Во-первых, списочный контекст должен быть обеспечен «окружением». В предшествующем примере за это отвечает списочное присваивание. Ранее мы видели, что такой контекст предоставляет оператор цикла foreach. Оператор print также предоставляет его. Но вам не придется это заучивать.

Посмотрев на синтаксические сводки, разбросанные по оставшейся части книги, можно увидеть операторы, в определении которых в качестве принимаемого аргумента указан *LIST*. Это и есть операторы, предоставляющие списочный контекст. Всюду в этой книге *LIST* используется как специфический технический термин, означающий «синтаксическая конструкция, предоставляющая списочный контекст». Например, найдя sort, вы обнаружите такую синтаксическую сводку:

```
sort LIST
```

Это значит, что функция sort предоставляет списочный контекст для своих аргументов.

Во-вторых, во время компиляции (когда Perl производит синтаксический анализ программы и транслирует ее во внутренние коды операций) каждый оператор, принимающий *LIST*, предоставляет списочный контекст для каждого синтаксического элемента этого *LIST*. Поэтому каждому оператору верхнего уровня или объекту в *LIST* известно на этапе компиляции, что от него ожидается создание самого лучшего списка, на который он способен. Это означает, что если сказать:

```
sort @dudes, @chicks, other();
```

то каждый из элементов, @dudes, @chicks и other(), знает на этапе компиляции, что от него ждут список, а не скалярную величину. Для отражения этого компилятор генерирует соответствующие внутренние коды операций.

Позже, на этапе выполнения (когда внутренние коды операций интерпретируются фактически), каждый из этих элементов в *LIST* порождает свой список, а затем (это важно) все отдельные списки объединяются вместе, друг за другом, в один общий список. Этот расплющенный одномерный список и передается в конечном итоге функции, которая требовала *LIST*. Поэтому, если @dudes содержит (Fred,Barney), @chicks содержит (Wilma,Betty), а функция other() возвращает список из одного элемента (Dino), то *LIST*, который получит sort, будет таким:

```
(Fred,Barney,Wilma,Betty,Dino)
```

а *LIST*, который sort вернет, будет следующим:

```
(Barney,Betty,Dino,Fred,Wilma)
```

Одни операторы порождают списки (например, keys), другие потребляют их (например, print), а третьи преобразуют один список в другой (например, sort). Операторы последней категории можно рассматривать как фильтры, но в них, в отличие от оболочки командного интерпретатора, поток данных идет справа налево, поскольку списочные операторы воздействуют на аргументы, передаваемые им справа. Можно расположить подряд несколько списочных операторов:

```
print reverse sort map {lc} keys %hash;
```

В этом выражении функция keys получает ключи из хеша %hash и возвращает их функции map, которая переводит их все в нижний регистр, применяя для каждого оператор lc, и передает затем функции sort, которая сортирует их и передает

функции reverse, которая обращает порядок элементов списка и передает их функции print, которая их выводит.

Как видите, описать это на Perl куда проще, чем на естественном языке.

Есть много других ситуаций, когда обработка списка порождает более естественный код. Невозможно все их здесь перечислить, но для примера вернемся на минуту к регулярным выражениям. Мы уже обсуждали применение шаблонов для поиска в скалярном контексте, но если использовать шаблон в списочном контексте, происходит нечто иное: все ссылки на найденный текст вытаскиваются в виде списка. Допустим, вы производите поиск в файле журнала или почтовом ящике и хотите разобрать строку, содержащую время в формате «12:59:59 am». Можно сделать это так:

```
my ($hour, $min, $sec, $ampm) = /(\d+):(\d+):(\d+) *(\w+)/;
```

Это удобный способ присвоить значения одновременно нескольким переменным. Но ничуть не сложнее сказать:

```
my @hmsa = /(\d+):(\d+):(\d+) *(\w+)/;
```

и поместить все четыре значения в один массив. Как ни странно, но, отделяя мощь регулярных выражений от мощи выражений Perl, списочный контекст увеличивает мощь языка. Хотя мы редко признаемся в этом, Perl является не только диагональным языком, но и фактически ортогональным языком. Можно один пирог съесть дважды.

Чего вы не знаете, то вам (сильно) не навредит

В заключение позвольте еще раз вернуться к концепции Perl как естественного языка. Говорящие на естественном языке могут иметь разный уровень мастерства, предпочитать какие-то подмножества языка, учить язык по ходу дела и обычно с пользой применять язык, не дожидаясь того времени, когда изучат его целиком. Вы пока не знаете Perl целиком, как не знаете целиком и тот язык, на котором разговариваете. Но в культуре Perl это официально считается нормальным. Вы можете извлекать пользу из работы с Perl, даже если мы еще не рассказали вам, как создавать собственные подпрограммы. Мы едва коснулись вопросов применения Perl для системного администрирования, быстрого создания прототипов, сетевого взаимодействия или в качестве объектно-ориентированного языка. Об этих вопросах можно написать целые главы. (Если вдуматься, то мы это уже сделали.)

Но в конечном итоге читатель должен выработать собственное отношение к Perl. Как художник, вы имеете право на собственные муки творчества. Мы можем показать, как *мы* пишем картины, но не можем указывать, как писать картины *вам*. Есть Несколько Способов Сделать Это.

Получите причитающееся вам удовольствие.

II

Анатомия Perl

2

Всякая всячина

Мы начнем с малого и посвятим эту главу элементам Perl.

Поскольку мы начинаем с малого, продвижение по следующим главам будет вынужденно происходить от малого к большому. Это значит, что мы выбрали восходящий принцип и, начав с мельчайших составляющих программ Perl, будем собирать из них более сложные конструкции, подобно тому, как молекулы составляются из атомов. Недостаток такого подхода в том, что из-за деревьев можно не увидеть леса, потерявшись в нагромождении подробностей. Преимущество же заключается в том, что вам будут понятны примеры, приводимые по ходу изложения. (Если вы предпочитаете нисходящий порядок, переверните книгу и читайте главы в обратном порядке.)

Каждая глава базируется на материале предшествующей главы (или *последующей*, если вы читаете с конца), поэтому будьте осторожны, если у вас есть привычка перескакивать через страницы.

Разумеется, мы только приветствуем, если вы станете подглядывать в справочные материалы в конце книги. (Это не считается перескакиванием через страницы.) В частности, отдельные слова, оформленные `моноширинным шрифтом`, вы, скорее всего, найдете в главе 27. Мы старались сделать изложение не зависящим от операционной системы, но, если вы не знакомы с терминологией UNIX и столкнетесь со словом, как будто имеющим иной смысл, чем вы ожидали, следует поискать его в глоссарии. Если не поможет глоссарий, то, вероятно, поможет алфавитный указатель. Если и это не поможет, воспользуйтесь своей любимой поисковой системой.

Атомы

За кулисами происходит множество незаметных вещей, и вскоре мы о них расскажем, однако самыми маленькими объектами, с которыми обычно приходится работать в Perl, являются отдельные символы. Мы говорим именно о символах: в прежние времена Perl легко путал байты с символами, а символы с байтами, но в наступившую эпоху глобальных сетей нужно тщательно различать эти две вещи.

Можно, конечно, писать на Perl, ориентируясь исключительно на 7-разрядный набор символов ASCII. По историческим причинам Perl считает, что байты в диапазоне 128–255 относятся к набору символов ISO-8859-1 (Latin1), где их коды соответствуют кодам тех же символов Юникода. Чтобы сообщить интерпретатору Perl, что текущий файл с исходным программным кодом должен интерпретироваться как текст Юникода в кодировке UTF-8, поместите следующее объявление в начало файла:

```
use utf8;
```

Как будет сказано в главе 6, поддержка Юникода появилась в Perl еще в прошлом тысячелетии. Эта поддержка внедрена в язык повсеместно: можно использовать символы Юникода как в идентификаторах (например, именах переменных), так и в строковых литералах. В случае применения Юникода не нужно беспокоиться о том, сколько битов или байтов занимает символ. Perl представляет дело так, как если бы все символы Юникода имели одинаковый размер (а именно 1), даже внутреннее представление того или иного символа требует нескольких байтов. Обычно Perl использует для внутреннего представления UTF-8, кодировку с переменной длиной. (Например, «смайлик» ☺ в кодировке Юникода, U+263A, внутренне представлен последовательностью из трех символов, но можно не беспокоиться об этом.)

Если вы позволите несколько развить нашу аналогию с физическими элементами, то символы атомарны в том же самом смысле, что и атомы различных элементов. Да, они состоят из более мелких частиц, которые называются битами и байтами, но если расколоть символ на части (разумеется, в ускорителе символов), то отдельные биты и байты утратят отличительные «химические» свойства целого символа. Как нейтроны являются деталями реализации атома урана-238, так и байты являются деталями реализации символа U+263A.

Так что не беспокойтесь о мелочах. Переходим к более крупным и приятным вещам.

Молекулы

Perl является языком *свободной формы* (*free-form language*), но это не значит, что код на Perl вообще бесформенный. Компьютерщики обычно используют этот термин для обозначения языка, в котором пробелы, символы табуляции и перевода строки можно помещать где угодно – за исключением тех мест, где этого делать нельзя.

Первое очевидное место, в котором нельзя ставить пробельные символы, – это внутри лексемы. *Лексема* (*token*) – это последовательность символов с единицей смыслового содержания, примерно как слово в естественном языке. Но, в отличие от обычных слов, лексема может содержать символы, не являющиеся буквами, и требуется лишь, чтобы они сплоченно составляли смысловую единицу. (В этом отношении лексемы более похожи на молекулы, которые не обязательно состоят из атомов только одного вида.) Например, числа и математические операторы считаются лексемами. *Идентификатор* (*identifier*) – это лексема, которая начинается с буквы или соединительного знака (такого как символ подчеркивания) и содержит только буквы, комбинируемые знаки[1], цифры и символы под-

[1] К ним относятся, например, диакритические знаки. – *Прим. ред.*

черкивания. Лексема не может содержать пробельных символов, поскольку в результате она оказалась бы разбитой на две лексемы; так же пробел в обычном слове превращает его в два слова.[1]

Пробельный символ допускается помещать между любыми двумя лексемами, но *обязательным* его присутствие является только между такими лексемами, которые без него могут быть приняты за одну. Для этой цели могут использоваться любые символы из числа пробельных. Символы перевода строки не эквивалентны пробелам и символам табуляции только внутри строк, заключенных в кавычки, внутри форматов и некоторых других последовательностей строчно-ориентированной маскировки. Конкретно символ новой строки не служит концом оператора, как в некоторых других языках (например, FORTRAN или Python). В Perl завершению оператора служит точкой с запятой, как в C и производных языках вроде C++ и Java.

Пробельные символы Юникода допускается использовать в программах на Perl, но при этом следует проявлять осторожность. Применяя особые разделители абзацев и строк из набора символов Юникода, следует учитывать, что Perl может пронумеровать строки не так, как это сделает ваш текстовый редактор, из-за чего может усложниться толкование сообщений об ошибках. Лучше придерживаться использования старых добрых символов перевода строки.

Лексемы распознаются путем жадного поиска: если в некотором месте анализатор Perl имеет выбор между короткой и длинной лексемами, он предпочтет длинную. Если вам нужно, чтобы эта лексема считалась за две, вставьте пробельный символ в соответствующей точке. (Мы в любом случае стараемся окружать дополнительными пробелами большинство инфиксных операторов, чтобы повысить удобочитаемость.)

Комментарии начинаются с символа # и простираются до конца строки. В отношении разделения лексем комментарий считается за пробельный символ. В языке Perl содержимое комментария, каким бы оно ни было, не наделяется никаким особым смыслом:[2]

```
my $comet = 'Haley              # Это - комментарий
```

Еще одна особенность связана с тем, что, если строка начинается символом = и в этом месте допустим оператор, Perl игнорирует все, начиная с этой строки вплоть до строки, начинающейся последовательностью символов =cut. Игнорируемый текст считается *документацией*, или *pod* («plain old documentation»). В дистрибутив Perl входят программы, извлекающие комментарии типа pod из модулей Perl и преобразующие их в простой текст, страницы руководства, документы LaTeX, HTML или XML. Анализатор Perl, напротив, извлекает код Perl из модулей Perl и игнорирует pod. Можно рассматривать это как альтернативу, спо-

[1] Проницательный читатель отметит, что строковые литералы могут содержать пробельные символы. Но это возможно лишь благодаря двойным кавычкам, которые не дают пробелам сбежать.

[2] Тут мы немножко приврали. Анализатор Perl все же ищет ключи командной строки в начальной строке #! (см. главу 17). Он может также интерпретировать директивы номера строки, созданные препроцессором (см. раздел «Генерация Perl из других языков» главы 21). Отдельные модули, такие как Perl::Critic и Smart::Comments, также используют специальные комментарии, чтобы выяснить, что требуется сделать.

соб создания многострочных комментариев. Программный код в таком разделе pod даже не компилируется:

```
=pod

my $dog = 'Spot';
my $cat = 'Buster';

=cut
```

Можете не верить, но для модулей Perl, документированных таким способом, документация никогда не теряется. О деталях документирования программ рассказывается в главе 23; там же описывается обработка многострочных комментариев средствами Perl.

Не стоит пренебрегать и обычным символом комментария. Есть что-то умиротворяющее в аккуратной колонке символов # на левой границе многострочного комментария. Она немедленно сообщает взгляду: «Это не код». Заметьте, что даже в языках, имеющих средства создания многострочных комментариев, таких как C, программисты часто все равно помещают колонку символов * по левому краю комментариев. Внешний вид часто важнее, чем кажется по внешнему виду:

```
# начало многострочного комментария
# my $dog = 'Spot';
# my $cat = 'Buster';
```

Подобно химии и естественному языку, Perl позволяет строить все более и более сложные структуры из простых. Мы уже употребляли слово *оператор* (*statement*): это просто последовательность лексем, составляющих вместе команду, т.е. предложение в повелительном наклонении. Последовательность команд можно объединить в *блок*, который заключается в *фигурные скобки* (*braces*), любовно называемые также «завитушками» теми, кто считает, что braces – это подтяжки.[1] Блоки, в свою очередь, могут объединяться в еще более крупные блоки. Некоторые блоки действуют как *подпрограммы* (*subroutines*), которые могут объединяться в *модули* (*modules*), которые могут объединяться в *программы* (*programs*). Но мы слишком забегаем вперед – это темы последующих глав. Лучше построим из символов еще несколько лексем.

Встроенные типы данных

Прежде чем начать разговор о различных типах лексем, которые можно построить из символов, требуется ввести еще несколько абстракций. Точнее, нам потребуется три типа данных.

Языки программирования отличаются числом и видами имеющихся в них типов данных. В отличие от некоторых широко распространенных языков, предлагающих множество запутанных типов для сходных видов значений, в Perl довольно мало встроенных типов данных. Возьмите C, где можно встретить `char`, `short`, `int`, `long`, `long long`, `bool`, `wchar_t`, `size_t`, `off_t`, `regex_t`, `uid_t`, `u_longlong_t`, `pthread_key_t`, `fp_exception_field_type` и прочие. И это далеко не исчерпывающий список целочисленных типов! А помимо целых есть еще числа с плавающей запятой, а также указатели, а также строки.

---

[1] Другое значение слова braces. – *Прим. перев.*

Всем этим сложным типам данных в Perl соответствует единственный тип: скаляр. (Как правило, потребности разработчика удовлетворяются простыми типами данных Perl, но если потребуется что-то более сложное, вы вольны конструировать произвольные динамические типы с использованием объектно-ориентированных возможностей Perl – см. главу 12.) Основными тремя типами данных Perl являются *скаляры*, *массивы скаляров* и *хеши скаляров* (называемые также *ассоциативными массивами*). Некоторые предпочитают называть их не типами, а *структурами данных*. Мы не возражаем.

Скаляры являются базовым типом, на основе которого строятся более сложные структуры. Скаляр хранит одно простое значение, обычно строку или число. Элементы этого простого типа могут объединяться в один из двух составных типов. *Массив* – это упорядоченный список скаляров, обращение к которым происходит с помощью целочисленного индекса (subscript). В Perl индексация начинается с нуля. Однако, в отличие от многих языков программирования, Perl считает допустимыми отрицательные индексы, которые ведут обратный отсчет с конца индексируемой структуры. (Это относится как к обычному индексированию, так и к различным операциям с подстроками и подсписками.) Напротив, *хеш* является неупорядоченным списком пар ключ/значение, обращение к которым производится путем использования строки (*ключа*) в качестве индекса для поиска скаляра (*значения*), соответствующего данному ключу. Переменные всегда имеют один из этих трех типов. Помимо переменных в Perl есть другие абстракции, которые можно считать типами данных: например, дескрипторы файлов, дескрипторы каталогов, форматы, подпрограммы, таблицы имен и записи таблиц имен.

Абстракции – это чудесно, и по ходу дела у нас их наберется много, но в некотором отношении они бесполезны. Непосредственно с абстракцией делать ничего нельзя. Поэтому в языках программирования имеется синтаксис. Мы должны познакомить вас с некоторыми видами синтаксических термов, которые можно использовать для объединения абстрактных данных в выражения. Нам нравится применять технический термин *терм* при необходимости поговорить в терминах синтаксических единиц. (Хм, похоже, это может создать терминологическую путаницу. Вспомните просто, что школьный учитель математики говорил о *термах* (членах) в уравнении, и вы не сильно ошибетесь.)

Подобно членам математического уравнения, назначением термов в Perl является создание величин, с которыми работают операторы типа сложения и умножения. Однако, в отличие от математического уравнения, Perl должен что-то делать с величинами, которые он вычисляет, а не просто размышлять с карандашом в руке, равны ли между собой две части уравнения. Действие чаще всего состоит в том, чтобы где-то запомнить значение:

```
$x = $y;
```

Это пример оператора присваивания (а не оператора равенства чисел, который в Perl записывается как ==). Присваивание получает значение из `$y` и помещает его в `$x`. Обратите внимание, что терм `$x` используется не ради его значения, а ради его адреса. (Прежнее значение `$x` будет уничтожено присваиванием.) Мы говорим, что `$x` является *l-значением* (от left value – левое значение), подразумевая под этим, что данный тип памяти может фигурировать в левой части присваивания. Мы говорим, что `$y` является *r-значением* (от right value – правое значение), поскольку оно расположено в правой части.

Есть и третий тип значений, называемый *временными значениями*, который нужно понять, если вы и впрямь хотите узнать, что в действительности делает Perl с вашими l-значением и r-значением. Если мы выполняем какие-то осмысленные математические действия и говорим:

```
$x = $y + 1
```

то Perl берет правое значение $y и прибавляет к нему правое значение 1, в результате чего создается временное значение, которое, в конечном итоге, присваивается левому значению $x. Возможно, вам будет легче представить себе зрительно, что происходит, если мы сообщим, что Perl запоминает эти временные значения во внутренней структуре, называемой *стеком*.[1] Термы выражения (о которых мы говорим в данной главе) стремятся протолкнуть значения в стек, тогда как операторы выражения (о которых мы будем говорить в следующей главе) стремятся вытолкнуть их обратно. В некоторых случаях при этом в стеке остается временный результат, с которым будет работать следующий оператор. Все эти проталкивания и выталкивания уравновешиваются: к тому времени, когда вычисление выражения заканчивается, стек становится совершенно пуст (или столь же пуст, как в начале вычисления). Мы еще поговорим о временных значениях. Некоторые термы могут быть только правыми значениями, как, скажем, 1 (единица) в примере выше, тогда как другие могут быть и левыми, и правыми значениями. В частности, как показывают вышеприведенные присваивания, переменная способна выполнять и ту и другую роль. Этому и посвящен следующий раздел.

Переменные

Неудивительно, что существует три типа переменных, соответствующие трем описанным выше абстрактным типам данных. Каждой из таких переменных предшествует так называемый разыменовывающий префикс (sigil).[2] Имя скалярной переменной всегда начинается символом $, даже если речь о скаляре, являющемся частью массива или хеша. Действие этого символа несколько напоминает использование определенного артикля «the» в английском языке. См. табл. 2.1.

Таблица 2.1. Доступ к скалярным значениям

Конструкция	Значение
$days	Простое скалярное значение $days
$days[28]	29-й элемент массива @days
$days{'Feb'}	Значение "Feb" из хеша %days

Обратите внимание, что одно и то же имя можно одновременно использовать для скаляра $days, массива @days и хеша %days, и Perl не запутается.

---

[1] Стек действует подобно подпружиненному устройству раздачи тарелок в буфетах – можно *проталкивать* (*push*) тарелки на вершину стека (стопки), а можно *выталкивать* их (pop) обратно (если воспользоваться профессиональным жаргоном вычислительных наук).

[2] По-видимому, потому, что берет обычное имя и делает его необычным.

Есть и другие, необычные скалярные термы, применяемые в особых ситуациях, в которые мы пока вникать не будем. Они перечислены в табл. 2.2.

Таблица 2.2. Синтаксис скалярных термов

Конструкция	Значение
${days}	То же, что $days, но устраняется неоднозначность перед буквенно-цифровыми символами
$Dog::days	Еще одна переменная $days, в пакете Dog
$#days	Последний индекс массива @days
$days->[28]	29-й элемент массива, на который указывает ссылка $days
$days[0][2]	Многомерный массив
$days{2000}{'Feb'}	Многомерный хеш
$days{2000,'Feb'}	Эмуляция многомерного хеша

Массивы в целом (или *срезы* массивов и хешей) именуются с помощью разыменовывающего символа @, действующего во многом аналогично словам «эти» или «те». Их синтаксис описывается в табл. 2.3.

Таблица 2.3. Синтаксис списочных термов

Конструкция	Значение
@days	Массив, содержащий ($days[0], $days[1],... $days[n])
@days[3, 4, 5]	Срез массива, содержащий ($days[3], $days[4], $days[5])
@days[3..5]	Срез массива, содержащий ($days[3], $days[4], $days[5])
@days{'Jan','Feb'}	Срез хеша, содержащий ($days{'Jan'},$days{'Feb'})

Хеш в целом именуется с помощью символа %, как показано в табл. 2.4.

Таблица 2.4. Синтаксис скалярных термов

Конструкция	Значение
%days	(Jan => 31, Feb => $leap ? 29 28,)

Все эти конструкции могут также служить в качестве l-значений, задавая адрес, по которому можно присвоить значение. Для массивов, хешей и срезов массивов и хешей l-значение предоставляет местоположение для присваивания, что дает возможность присваивать несколько значений за одну операцию:

```
@days = 1    7:
```

Имена

Мы говорили о хранении значений в переменных, но и сами переменные (их имена и связанные с ними определения) тоже должны где-то храниться. Теория называет такие места *пространствами имен* (*namespaces*). Perl предлагает два вида пространств имен, которые часто называются *таблицами имен* (*symbol tables*)

и *областями лексической видимости* (*lexical scopes*).[1] Таблиц имен и областей лексической видимости может быть сколько угодно, и каждое определяемое вами имя хранится в той или иной из них. Далее мы рассмотрим оба типа пространств имен, а пока скажем лишь, что таблицы имен являются глобальными хешами, содержащими записи таблицы имен для глобальных переменных (в том числе хешей для других таблиц имен). Напротив, лексические области видимости являются *анонимной* временной памятью, находящейся не в какой-либо таблице имен, а прикрепленной к блоку кода вашей программы. Они содержат переменные, видимые только в данном блоке. (Это мы и подразумеваем под *областью видимости*. Слово *лексическая* означает, что она относится к тексту, и это совсем не то, что подразумевалось бы лексикографом. Не вините нас.)

В каждом пространстве имен (глобальном или лексическом) переменным каждого типа отведено собственное подпространство имен, определяемое разыменовывающим префиксом. Можно, не опасаясь конфликта, использовать одно и то же имя для скалярной переменной, массива или хеша (а также дескриптора файла, подпрограммы, метки или любимой ручной ламы). Это значит, что `$foo` и `@foo` суть две различные переменные. Согласно предшествующим правилам, это также значит, что `$foo[1]` является элементом `@foo`, никак не связанным со скалярной переменной `$foo`. Это может показаться странным, и правильно, поскольку это *действительно* странно[2].

Имена подпрограмм могут начинаться с символа `&`, хотя при вызове подпрограмм указание разыменовывающего префикса не обязательно. Обычно подпрограммы не считаются l-значениями, но в Perl разрешается возвращать из подпрограммы l-значение и выполнять присваивание по отношению к нему, что выглядит как присваивание подпрограмме.

Иногда требуется имя для «всего, что названо foo», независимо от того, какие разыменовывающие префиксы использовались. Поэтому имена записей таблицы имен могут начинаться символом `*`; при этом звездочка обозначает все остальные разыменовывающие префиксы. Эти записи называются *записями глобальной таблицы имен* (*typeglobs*), и существует несколько способов их применить. Они могут также выступать в качестве l-значений. Посредством присваивания типу данных typeglob в Perl реализуется импорт имен из одной таблицы имен в другую. Мы еще расскажем об этом подробнее.

Подобно большинству языков программирования, Perl резервирует список слов, которые считаются в нем особыми ключевыми словами. Однако поскольку имена переменных всегда начинаются с разыменовывающего префикса, зарезервированные слова фактически не конфликтуют с именами переменных. Но некоторые другие типы имен не имеют разыменовывающих префиксов – как, например, дескрипторы файлов и метки. Что касается таких имен, следует беспокоиться (хотя и не сильно) о возможном конфликте с зарезервированными словами. Поскольку в большинстве зарезервированных слов применяются только символы в нижнем регистре, рекомендуем выбирать имена меток и дескрипторов файлов так, чтобы

---

[1] В конкретных реализациях Perl они также называются *пакетами* (*packages*) и *панелями* (*pads*), но приведенные нами более длинные названия являются общепринятыми терминами в данной области, поэтому мы стараемся придерживаться их. Извините.

[2] Это настолько странно, что в Perl 6 мы решили сделать все наоборот, что тоже по-своему странно.

они содержали заглавные буквы. Например, предпочтя open(LOG, logfile) опасному open(log, "logfile"), вы не введете Perl в заблуждение, будто имеется в виду встроенный оператор log (который, конечно же, вычисляет логарифмы). Использование символов в верхнем регистре для имен дескрипторов файлов также повышает удобочитаемость[1] и предотвращает конфликты с зарезервированными словами, которые могут появиться в будущем. По сходным причинам в именах модулей, созданных пользователем, первую букву обычно делают заглавной: так их легче отличать от встроенных модулей, которые называются прагмами и именуются только в нижнем регистре. А когда мы доберемся до объектно-ориентированного программирования, вы увидите, что имена классов обычно начинают с заглавных все по той же причине.

Как можно было заключить из предыдущего абзаца, регистр символов в идентификаторах имеет значение – FOO, Foo и foo являются в Perl разными именами. Идентификаторы начинаются с символа подчеркивания или буквы, могут иметь любую длину (где «любая» длина находится в диапазоне от 1 до 251 включительно) и могут содержать буквы, цифры и символы подчеркивания. Если вы объявили (use utf8), что исходный программный код является текстом Юникода, правила игры немного меняются: теперь идентификаторы должны начинаться соединительным знаком (такого как символ подчеркивания) или любым символом Юникода, обладающим свойством XID_Start (XIDS), за которым может следовать любой символ со свойством XID_Continue (XIDC). Это открывает нам свыше 100 000 различных символов[2], которые можно использовать в идентификаторах, в том числе идеограммы, считающиеся буквами, но мы не рекомендуем использовать их, если вы не умеете читать их.[3] См. главу 6.

Имена с разыменовывающими префиксами не обязаны, строго говоря, быть идентификаторами. Они могут начинаться с цифр, и тогда должны содержать только цифры – как, например, имя $123. Имена, начинающиеся с символа, отличного от буквы, цифры или соединительного знака, ограничены (обычно) одним символом (например, $? или $$) и, как правило, имеют в Perl предопределенное значение. Например, как и в оболочке UNIX, $$ является идентификатором текущего процесса, а $? – кодом завершения последнего дочернего процесса. В Perl также имеется открытый синтаксис для внутренних имен переменных. Всякая переменная вида ${^*NAME*} является специальной переменной, зарезервированной для использования интерпретатором Perl. Все эти имена, не являющиеся идентификаторами, принудительно помещаются в главную таблицу имен. Некоторые примеры можно найти в главе 25.

Можно ошибочно прийти к выводу, что имена и идентификаторы – это одно и то же, но под *именем* мы обычно подразумеваем *полностью квалифицированное имя*, т.е. имя, содержащее название своей таблицы имен. Такое имя может быть образовано последовательностью идентификаторов, разделяемых лексемой ::

---

1 Один из принципов проектирования Perl заключается в том, что разные вещи должны и выглядеть по-разному. Сравните это с языками, в которых делается попытка заставить разные вещи выглядеть одинаково в ущерб удобочитаемости.

2 В версии Юникода v6.0.

3 Версия Perl v5.14 не нормализует имена переменных, поэтому даже при внешней похожести идентификаторы могут оказаться разными, если в одном используются комбинированные символы, а в другом – составные.

```
$Santa::Helper::Reindeer::Rudolph::nose
```

Это похоже на каталоги и имена файлов в путях файловой системы:

```
/Santa/Helper/Reindeer/Rudolph/nose
```

В версии этого обозначения, используемой Perl, все идентификаторы, кроме последнего, являются именами вложенных таблиц имен, тогда как последний является именем переменной в самой глубоко вложенной таблице имен. Например, в приведенной ранее переменной именем таблицы является Santa::Helper::Reindeer::Rudolph::, а фактическая переменная в этой таблице имен – $nose (нос). (А значением этой переменной является, конечно, «красный».)

Таблицу имен в Perl называют также *пакетом* (*package*), поэтому соответствующие переменные часто называются пакетными. Пакетные переменные номинально принадлежат пакету, в котором объявлены, но являются глобальными в том же смысле, в каком глобальны сами пакеты. Это значит, что каждый может посредством имени пакета добраться и до переменной, просто по ошибке это сделать трудно. Например, всякая программа, в которой упоминается $Dog::bert, обращается к переменной $bert из пакета Dog::. А это переменная совсем иначе, чем $Cat::bert. См. главу 10.

Переменные, относящиеся к лексической области видимости, не принадлежат никакому пакету, поэтому имена переменных с лексической областью видимости не могут содержать последовательность символов ::. (Переменные с лексической областью видимости объявляются с помощью ключевого слова my, our или state.)

Поиск имен

Итак, вопрос: что в имени тебе моем? Как Perl определяет, что вы имеете в виду, обращаясь к переменной с именем $bert? Хороший вопрос. Вот правила, которыми руководствуется анализатор Perl, пытаясь разобрать неквалифицированное имя в контексте:

1. Сначала Perl просматривает ближайший охватывающий блок и проверяет, нет ли в этом блоке данной переменной в объявлениях my, our или state (почитайте о них в главе 27 и разделе «Объявления с областью видимости» главы 4). Если объявление my или state найдено, то переменная имеет лексическую область видимости и не принадлежит какому-либо пакету – она существует только в этой лексической области видимости (иначе говоря, во временной памяти этого блока). Поскольку лексические области видимости безымянны, никто вне этого фрагмента кода не может даже упомянуть вашу переменную.[1]
2. Если переменная не найдена в непосредственно охватывающем блоке, Perl ищет блок, охватывающий данный, и переменную с лексической областью

[1] При использовании объявления our вместо my или state создается *псевдоним* с лексической областью видимости для пакетной переменной, а не действительная переменная с лексической областью видимости, как происходит в случае объявления my или state. Сторонний код может получить доступ к реальной переменной через ее пакет. но во всех остальных отношениях объявление our ведет себя как объявление my. Это удобно, если вы пытаетесь ограничить свое собственное использование глобальных переменных с помощью прагмы strict (см. описание strict в главе 5). Но всегда предпочтительнее использовать my или state, если вам не нужна глобальная переменная.

видимости в этом более широком блоке. В случае успеха Perl считает, что переменная принадлежит только лексической области видимости от точки объявления до конца блока, в котором она объявлена, включая вложенные блоки, например тот, из которого мы пришли на шаге 1. Если Perl не находит такое объявление, он повторяет шаг 2, пока не кончатся охватывающие блоки.

3. Когда закончились охватывающие блоки, Perl рассматривает единицу компиляции целиком – и ищет в ней объявления, как если бы она была блоком (*единицей компиляции* является весь текущий файл или строка, компилируемая в данный момент оператором `eval` *STRING*). Если единицей компиляции является файл, то это самая большая лексическая область видимости из возможных, и Perl прекращает дальнейший поиск лексических областей видимости, поэтому мы переходим к шагу 4. Если же единицей компиляцией является строка, то все не так просто. Строка кода Perl, компилируемая на этапе выполнения программы, притворяется блоком в лексической области видимости, из которой вызван `eval` *STRING*, хотя действительными границами лексической области видимости являются не фигурные скобки, а границы строки, содержащей код. Поэтому, если Perl не находит переменную в лексической области видимости строки, дальнейшие действия исходят из того, что `eval` *STRING* является блоком, поэтому выполняется возврат к шагу 2, только на этот раз поиск начинается с лексической области видимости оператора `eval` *STRING*, а не с области внутри его строки.

4. Если мы попали сюда, это значит, что Perl не нашел для переменной никаких объявлений (ни `my`, ни `our`). Теперь Perl отказывается от поиска переменной с лексической областью видимости и считает, что переменная является пакетной. Если действует директива `strict`, то в данном возникнет ошибка, если только эта переменная не является одной из предопределенных переменных Perl или не импортирована в текущий пакет. Дело в том, что данная директива запрещает использование неквалифицированных глобальных имен. Однако мы еще не покончили с лексическими областями видимости. Perl осуществляет такой же поиск лексических областей видимости, как на шагах с 1 по 3, но на этот раз ищет не объявления переменных, а объявления пакетов. Если он находит такое объявление пакета, то полагает, что текущий код компилируется для указанного пакета, и делает имя объявленного пакета префиксом переменной.

5. Не обнаружив объявления пакета в окружающей лексической области видимости, Perl ищет имя переменной в неименованном пакете верхнего уровня, именем которого оказывается `main`, если только у него не отсутствует тег имени. Поэтому, в отсутствие противоречащих тому объявлений, `$bert` означает то же, что `$::bert`, что означает то же, что `$main::bert`. (Но поскольку `main` является пакетом в неименованном пакете верхнего уровня, то это также `$::main::bert`, и `$main::main::bert`, `$::main::main::bert` и т.д. Это можно считать бесполезным фактом. Однако загляните в раздел «Таблицы имен» главы 10.)

Есть несколько следствий из этих правил поиска, которые могут быть неочевидными, поэтому сформулируем их явно.

- Поскольку файл является наибольшей лексической областью видимости из возможных, переменную с лексической областью видимости нельзя увидеть вне файла, в котором она определена. Файловые области видимости не могут быть вложенными.

- Любой фрагмент Perl компилируется, по крайней мере, в одной лексической области видимости и ровно в одной области видимости пакета. Обязательной лексической областью видимости является, конечно, сам файл. Дополнительные лексические области видимости образуют все охватывающие блоки. Весь код Perl компилируется также в области видимости ровно одного пакета, и хотя объявление любого пакета имеет лексическую область видимости, сами пакеты не ограничены лексически, т.е. являются глобальными.
- Отсюда следует, что поиск неквалифицированного имени переменной может производиться во многих лексических областях видимости, но только в одной области видимости пакета – той, которая активна в данный момент (какая именно, определяется лексически).
- Имя переменной может быть связано только с одной областью видимости. Хотя в любом месте программы действуют, по крайней мере, две разные области видимости (лексическая и пакета), переменная может существовать только в одной из этих областей видимости.
- Отсюда следует, что неквалифицированное имя переменной может быть разрешено только в один адрес памяти: либо в первой охватывающей лексической области видимости, где оно объявлено, либо в текущем пакете – но там и там одновременно. Поиск прекращается, как только найден адрес памяти, а все адреса памяти, которые могли быть найдены при продолжении поиска, оказываются в результате скрытыми.
- Местонахождение обычного имени переменной можно однозначно определить на этапе компиляции.

Теперь вы все знаете о том, как компилятор Perl работает с именами, но иногда возникает такая сложность, что на этапе компиляции нужное имя оказывается *не известно*. Иногда требуется обратиться к чему-либо косвенным образом; мы называем это задачей *косвенного доступа*. Perl предоставляет для этого следующий механизм: буквенно-цифровое имя переменной можно заменить блоком, содержащим выражение, которое возвращает *ссылку* на действительные данные. Например, вместо

```
$bert
```

можно сказать:

```
${ some_expression() }
```

и если функция `some_expression()` возвращает ссылку на переменную `$bert` (или даже строку `"bert"`), то это равносильно тому, что вы сразу написали бы `$bert`. С другой стороны, если функция возвращает ссылку на `$ernie`, вы получите другую переменную. Продемонстрированный синтаксис является наиболее общим (и наименее доходчивым) вариантом косвенного доступа, но мы расскажем о некоторых его удобных разновидностях в главе 8.

Скалярные значения

Скаляр, будь он поименован прямо или косвенно, находись он в переменной, элементе массива или временной величине, всегда содержит единственное значение. Этим значением может быть число, строка или ссылка на другой фрагмент данных. Значение может даже отсутствовать, и в этом случае скаляр называют *неоп-*

ределенным (*undefined*). Хотя можно говорить, что скаляр «содержит» число или строку, скаляры не имеют типа: не требуется объявлять, что скаляр имеет тип целого числа или действительного числа, строки или иной.

В будущих версиях Perl, возможно, появятся объявления типов int, num и str, но не для того, чтобы осуществлять строгую типизацию, а чтобы помочь оптимизатору в случаях, когда он не справляется с задачей самостоятельно. Некоторые модули в CPAN уже используют эту возможность.

Строки в Perl – это последовательности символов без каких-либо искусственных ограничений на размер или содержимое. Проще говоря, не требуется решать заранее, насколько длинными будут строки, а включать в них можно любые символы, в том числе нулевые байты. Perl по возможности хранит числа как знаковые целые, а если это невозможно, то как величины с плавающей запятой и двойной точностью в родном формате машины. Точность значений с плавающей точкой не бесконечна. Об этом нужно помнить, иначе равенства типа (10/3 == 1/3*10) по загадочным причинам не будут выполняться.

Однако вы можете изменить представления Perl о числах посредством прагм bigint, bigrat и bignum. Они реализуют целые, рациональные (дробные) и вещественные числа произвольной точности. Это может приблизить фактические результаты арифметических операций к ожидаемым:

```
% perl -E 'say 10/3 == 1/3*10 ? "Yes" : "No"'
No

% perl -Mbigrat -E 'say 10/3 == 1/3*10 ? "Yes" : "No"'
Yes

% perl -E 'say 4/3 * 5/12'
0.555555555555555

% perl -Mbigrat -E 'say 4/3 * 5/12'
5/9
```

Чтобы получить доступ к этим более совершенным числам внутри программы (а не в командной строке), следует использовать объявления use bigint, use bigrat и use bignum:

```
use v5.14;
use bigrat;
say 1/3 * 6/5 * 5/4; # выводит "1/2"
```

При необходимости Perl осуществляет преобразование между различными подтипами, поэтому можно обращаться с числом как со строкой и со строкой как с числом, и Perl будет поступать как должно. Чтобы преобразовать строку в число, Perl внутренне использует нечто вроде функции *atof*(3) из библиотеки C. Чтобы преобразовать число в строку, он на большинстве машин осуществляет действие, эквивалентное вызову *sprintf*(3) с форматом "%.14g". Попытки преобразования нечисловой строки типа foo в число дают в результате числовой 0 и вызывают вывод предупреждающих сообщений, если он включен. Примеры определения типа данных, содержащихся в строке, можно найти в главе 5.

Хотя строки и числа почти всегда взаимозаменяемы, со ссылками дело обстоит несколько иначе. Ссылка – это строго типизованный, неприводимый указатель со встроенным подсчетом ссылок и вызовом деструктора. Это означает, что ссыл-

ки можно применять для создания сложных типов данных, в том числе определяемых пользователем объектов. При этом они все же остаются скалярами, потому что, какой бы сложной ни была структура данных, иногда требуется рассматривать ее как одиночное значение.

Под *неприводимостью* мы понимаем невозможность, например, преобразовать ссылку на массив в ссылку на хеш. Ссылки не преобразуются в другие типы указателей. Однако если использовать ссылку как число или строку, то получается числовое или строковое значение, которому обеспечено сохранение уникальности ссылки, даже несмотря на то, что «ссылочность» значения утрачивается при его копировании из действительной ссылки. Такие величины можно сравнивать или определять их тип. Но сделать что-либо еще с этими величинами трудно, поскольку нельзя преобразовать числа или строки обратно в ссылки. Обычно это не проблема, ведь Perl не только не принуждает заниматься арифметикой указателей. но и вообще не разрешает ее. Дополнительные сведения о ссылках см. в главе 8.

Числовые литералы

Числовые литералы задаются в одном из нескольких традиционных[1] форматов для целых и вещественных чисел:

```
my $x = 12345;          # целое
my $x = 12345.67;       # вещественное
my $x = 6.02e23,        # экспоненциальная запись
my $x = 4_294_967_296;  # подчеркивания для лучшей читаемости
my $x = 0377;           # восьмеричное
my $x = 0xffff;         # шестнадцатеричное
my $x = 0b1100_0000;    # двоичное
```

Поскольку Perl использует запятую в качестве разделителя в списках, она не может выступать в качестве разделителя групп разрядов в больших числах. Для этой цели Perl разрешает применять символ подчеркивания. Символ подчеркивания действует только в числовых литералах, определенных в тексте программы, но не в строках чисел или данных, прочитанных из любого другого источника. Аналогичным образом префиксы 0x для шестнадцатеричных чисел, префиксы 0b – для двоичных, и префиксы 0 – для восьмеричных действуют только в литералах. Автоматическое преобразование строки в число не распознает эти префиксы: необходимо выполнять явное преобразование[2] с помощью функции oct, которая обрабатывает также и шестнадцатеричные двоичные числа с префиксами 0x или 0b, соответственно.

Строковые литералы

Строковые литералы обычно заключаются в одинарные или двойные кавычки. Выбор кавычек дает во многом те же результаты, что в оболочке UNIX: для стро-

---

[1] Традиционных для UNIX. Если вы из другой культуры, добро пожаловать в нашу!

[2] Иногда программистам кажется, что Perl следует выполнять за них все необходимые преобразования входных данных. Но в реальной жизни слишком много начинающихся нулями десятичных чисел, чтобы Perl делал это автоматически. Например, почтовый индекс офиса O'Reilly Media в Кембридже, штат Массачусетс, выражается числом 02140. Почтовое ведомство оказалось бы в замешательстве, преобразуй ваша программа печати почтовых наклеек 02140 в десятичное 1120.

ковых литералов в двойных кавычках производится интерполяция символа обратной косой черты (\, backslash) и переменных, а для строк в одинарных кавычках – нет (за исключением \' и \\, что позволяет включать одинарные кавычки и обратную косую черту в строки, заключенные в одинарные кавычки). Если требуется встроить другую последовательность обратной косой черты, например, \n (перевод строки), то следует выбрать формат в двойных кавычках. (Последовательности с обратной косой чертой называются также *escape-последовательностями*, или *экранированными последовательностями*, поскольку позволяют временно «избежать» обычной интерпретации символов.)

Строка, заключенная в одинарные кавычки, должна отделяться пробелом от предшествующего слова, поскольку одинарная кавычка является допустимым, хотя и архаичным, символом идентификатора. Сегодня вместо него используется более удобочитаемая последовательность, ::. Это означает, что $main'var и $main::var суть одно и то же, но считается, что вторую запись легче читать и человеку, и программе.

Строки, заключенные в двойные кавычки, подвергаются различным видам интерполяции символов, многие из которых окажутся знакомы программирующим на других языках. Эти виды интерполяции перечислены в табл. 2.5.

Таблица 2.5. Экранированные последовательности с обратной косой чертой

Код	Значение
\n	Перевод строки (обычно LF)
\r	Возврат каретки (обычно CR)
\t	Горизонтальная табуляция
\f	Подача листа (form feed)
\b	Забой (backspace)
\a	Тревога (звонок)
\e	ESC-символ
\033	ESC в восьмеричном виде
\o{33}	Другое представление ESC в восьмеричном виде
\x7f	DEL в шестнадцатеричном виде
\x{263a}	Символ Юникода с кодом 0x263A
\N{LATIN SMALL LETTER E WITH ACUTE}	Символ с именем LATIN SMALL LETTER E WITH ACUTE, «é». Код символа в Юникоде: 0xE9.
\cN{ *U+E9* }	Снова символ 0xE9.
\cC	Control-C

Запись вида \N{НАЗВАНИЕ} применима лишь в сочетании с прагмой charnames, описанной в главе 29. Она позволяет указывать описательные названия символов – например, \N{GREEK SMALL LETTER SIGMA}, \N{greek:Sigma} или \N{sigma} – в зависимости от того, как вызвана прагма. Запись вида \N{U+ШЕСТНАДЦАТЕРИЧНЫЕ ЦИФРЫ} вызова прагмы charnames не требует, а кроме того, гарантирует, что для строки или регулярного выражения, в которое входит такая запись, будет применяться семантика Юникода. См. также главу 6.

Существуют также экранирующие последовательности, изменяющие регистр или «мета-свойства» последующих символов. Они сведены в табл. 2.6.

Таблица 2.6. Экранирующие последовательности трансляции

Код	Значение
\u	Перевести следующий символ в верхний регистр («регистр заголовка» (titlecase))[a]
\l	Перевести следующий символ в нижний регистр
\U	Перевести все следующие символы, вплоть до \E, в верхний регистр
\L	Перевести все следующие символы, вплоть до \E, в нижний регистр
\F	Принудительное приведение к единому регистру всех последующих символов, вплоть до \E
\Q	Добавить обратную косую черту перед всеми последующими символами, не являющимися буквами или цифрами[b], вплоть до \E
\E	Завершает действие символов \U, \L, \F и \Q

[a] Регистр заголовка (titlecase) – это регистр Юникода, по преимуществу соответствующий верхнему регистру. См. главу 6.

[b] Поддержка экранированной последовательности \F появилась в Perl v5.16. Единый регистр (foldcase map) – это особая форма символов, используемая для сравнения без учета регистра. См. главы 5 и 6.

Перевод строки можно непосредственно вставлять в строки исходного кода, т.е. строка кода может начинаться на одной строке экрана, а завершаться на другой. Часто это бывает удобно, но если вы забудете поставить закрывающую кавычку, сообщение об ошибке последует только тогда, когда Perl найдет очередную строку, содержащую символ кавычки, а это может произойти значительно позже в коде. К счастью, в той же строке обычно сразу возникает синтаксическая ошибка, а Perl в этом случае реагирует молниеносно и сообщает место, где, по его мнению, может начинаться незавершенная строка.

Интерполяции в строках, заключенных в двойные кавычки, подвергаются не только перечисленные выше экранированные последовательности с обратной косой чертой, но также скалярные и списочные значения. Это называется *интерполяцией переменных*. Иначе говоря, значения некоторых переменных можно вставлять непосредственно в строковые литералы. По сути, это просто удобная форма конкатенации строк.[1] Интерполяция переменных может выполняться для скалярных переменных, целых массивов (но не хешей), отдельных элементов массива или хеша, а также срезов (наборов индексов) массива или хеша. Интерполяция чего-либо другого не производится. Иными словами, интерполировать разрешено только выражения, начинающиеся символом $ или @, поскольку это те два символа (наряду с обратной косой чертой), которые ищет анализатор строк. Внутри строк литерал @, не являющийся частью идентификатора массива или среза, но предшествующий букве или цифре, должен предваряться обратной косой чертой (\@),

[1] Если включен вывод предупреждающих сообщений, Perl может сообщать об интерполяции в строки неопределенных значений как об операциях конкатенации и объединения, хотя вы не используете в этих местах такие операторы. Однако их создает за вас компилятор.

иначе возникает ошибка компиляции. Хотя целиковый хеш, обозначенный символом %, невозможно интерполировать в строку, отдельные значения хеша или его срезы допустимы, поскольку начинаются символами $ и @, соответственно.

Следующий фрагмент кода напечатает "Цена $100.":

```
my $Price = '$100',       # не интерполируется
print "Цена $Price.\n"; # интерполируется
```

Как и в некоторых оболочках командной строки, идентификатор можно заключить в фигурные скобки, чтобы разграничить его с примыкающими буквами или цифрами: "How ${verb}able!". Идентификатор в таких скобках преобразуется в строку, как всякий отдельностоящий идентификатор элемента в хеше. Например:

```
$days{'Feb'}
```

можно записать как:

```
$days{Feb}
```

и наличие кавычек будет подразумеваться. Более сложный текст в индексе интерпретируется как выражение и должен быть заключен в кавычки:

```
$days{'February 29th'} # Порядок.
$days{"February 29th"} # Тоже порядок. Интерполяция не требуется.
$days{ February 29th } # НЕВЕРНО, влечет ошибку анализатора.
```

В частности, всегда нужно использовать кавычки в таких срезах, как:

```
@days{'Jan','Feb'} # Порядок.
@days{"Jan","Feb"} # Тоже порядок
@days{ Jan, Feb }  # Не совсем (ошибка при use strict)
```

Если не принимать во внимание индексы интерполируемых переменных из массивов и хешей, интерполяция предлагает только один уровень вложенности. В противоположность ожиданиям людей, привыкших программировать в оболочках командных интерпретаторов, обратные кавычки (backticks) не приводят к интерполяции внутри двойных кавычек, и одинарные кавычки не препятствуют вычислению переменных, когда используются внутри двойных кавычек. Интерполяция является очень мощным средством, но в Perl находится под жестким контролем. Она происходит только в границах двойных кавычек и некоторых других подобных операций, о которых мы расскажем в следующем разделе:

```
print "\n"; # Порядок, вывод перевода строки.
print  \n ; # НЕВЕРНО, нет интерполирующего контекста.
```

Кавычки на любой вкус

Хотя обычно мы связываем кавычки с литеральными значениями, в Perl они действуют скорее как операторы, предоставляя различные возможности интерполяции и поиска по шаблону. Для этих режимов поведения Perl предлагает специальные символы вместо кавычек, а также более общий способ выбора символа кавычек для каждого из этих режимов. В табл. 2.7 показано, что вместо / можно указать любой буквенно-цифровой символ, отличный от пробельного. (Символы перевода строки и пробела больше нельзя использовать в качестве разделителей, хотя в доисторических версиях Perl это разрешалось.)

Таблица 2.7. Структуры «закавычивания»

Пользовательские	Общие	Значение	Интерполируется
'	q//	Строка-литерал	Нет
""	qq//	Строка-литерал	Да
`	qx//	Выполнение команды	Да
()	qw//	Список слов	Нет
//	m//	Поиск по шаблону	Да
s///	s///	Замена по шаблону	Да
tr///	y///	Транслирование символов	Нет
""	qr//	Регулярное выражение	Да

Некоторые из обозначений являются просто синтаксическими удобствами, избавляющими от набора многочисленных символов обратной косой черты в строках, заключенных в кавычки, особенно при поиске по шаблону, когда и без того легко запутаться в символах косой черты (slash) и обратной косой черты (backslash).

Если выбрать в качестве ограничителя одинарные кавычки, интерполяция переменных не производится даже в тех форматах, которые обычно интерполируются. Если левым ограничителем является открывающая круглая, квадратная, фигурная или угловая скобка, то правым ограничителем будет соответствующий закрывающий символ. (Внедренные ограничители должны иметь попарное соответствие.) Примеры:

```
my $single = q!Я сказал: "Ты сказал: 'Она сказала это'"!;

my $double = qq(Можем мы получить какую-нибудь "хорошую" $variable?);

my $chunk_of_code = q {
    if ($condition) {
        print "Попался!";
    }
};
```

Последний пример показывает, что можно использовать пробельный символ между спецификатором кавычек и левым ограничителем. В двухэлементных конструкциях, таких как s/// и tr///, когда первая пара кавычек служит парой скобок, вторая часть получает свой собственный символ начальной кавычки. В действительности вторая пара не обязательно должна совпадать с первой, поэтому можно писать такие вещи, как s<foo>(bar) или tr(a-f)[A-F]. Поскольку между двумя внутренними символами кавычек допускается наличие пробельного символа, последний пример можно записать даже так:

```
tr (a-f)
   [A-F];
```

Однако пробельные символы недопустимы, когда в качестве кавычек выступает символ #. q#foo# разбирается анализатором как строка 'foo', а q #foo# – как оператор кавычек q, за которым следует комментарий. Символ-ограничитель этого оператора будет взят со следующей строки. Комментарии можно помещать в середину двухэлементных конструкций, что позволяет писать:

```
s {foo}   # Заменить foo
  {bar};  #        на bar
tr [a-f]  # Перевести шестнадцатеричные цифры из нижнего регистра
   [A-F]; #                                    в верхний
```

Можно обойтись вообще без кавычек

Имя, для которого не нашлось иной интерпретации в грамматике языка, обрабатывается так, как если бы являлось строкой, заключенной в кавычки. Такие имена называют *голыми словами* (*barewords*).[1] Подобно дескрипторам файлов и меткам, голые слова, состоящие исключительно из букв нижнего регистра, могут вступать в конфликт со словами, которые будут зарезервированы в будущем. Если включен вывод предупреждений, Perl предостерегает о найденных голых слов. Например:

```
my @days = (Mon,Tue,Wed,Thu,Fri);
print STDOUT hello, ' ', world, "\n";
```

записывает в массив @days сокращенные названия дней недели и выводит "hello world" и символ перевода строки в STDOUT. Если опустить дескриптор файла, Perl попытается интерпретировать в качестве дескриптора файла hello, что приведет к синтаксической ошибке. Поскольку применение голых слов чревато возникновением ошибок, некоторые программисты склонны их избегать вовсе. Перечисленные выше операторы кавычек предоставляют много удобных форм, в том числе qw//, конструкцию «заключения слов в кавычки», которая точно заключает в кавычки список слов, разделенных пробелами:

```
my @days = qw(Mon Tue Wed Thu Fri);
print STDOUT "hello world\n";
```

Можно пойти и на то, чтобы вообще запретить использование голых слов. Если сказать:

```
use strict "subs";
```

то каждое голое слово приводит к возникновению ошибки времени компиляции. Это ограничение действует на протяжении всей охватывающей области видимости. Во вложенной области видимости можно отменить действие этой директивы, сказав:

```
no strict "subs"
```

Запрет использования голых слов – настолько хорошая идея, что, если сказать:

```
use v5.12;
```

или указать более новую версию, Perl включит все ограничения автоматически.

Обратите внимание, что голые идентификаторы в конструкциях типа:

```
"${verb}able"
$days{Feb}
```

---

[1] Имя переменной, дескриптор файлов, метка и тому подобное не считается за голое слово, поскольку назначение конструкции в этом случае определяется предшествующей или последующей лексемой (или же обеими). Предварительно объявленные имена, такие как имена подпрограмм, тоже не являются голыми словами. Голым слово является только в случае, когда анализатор понятия не имеет, что это такое.

не считаются голыми словами, поскольку явно разрешены, т.е. не относятся к случаю «отсутствия иной грамматической интерпретации».

Имя без кавычек, за которым следуют два двоеточия, например, main:: или Dog::, всегда считается именем пакета. На этапе компиляции Perl преобразует потенциально голое слово Camel:: в строку Camel, а стало быть, такое применение не вызовет осуждения.

Интерполяция массивов

Переменные, являющиеся массивами, интерполируются в строки, заключенные в двойные кавычки, путем слияния всех элементов массива. Разделитель при этом задается в переменной $"[1] (по умолчанию она содержит пробел). Следующие формы эквивалентны:

```
my $temp = join( $", @ARGV );
print $temp;

print "@ARGV";
```

В шаблонах поиска, где также выполняется интерполяция на манер двойных кавычек, возникает досадная двусмысленность: следует ли /$foo[bar]/ интерпретировать как /${foo}[bar]/ (где [bar] является классом символов в регулярном выражении) или как /${foo[bar]}/ (где [bar] является индексом массива @foo)? Если @foo не существует, то это, очевидно, класс символов. Если @foo существует, Perl пытается угадать, чем является [bar], и почти всегда успешно.[2] Если он угадывает неверно или вы страдаете паранойей, можно принудительно задать правильную интерпретацию с помощью фигурных скобок, как показано выше. Даже если вы всего лишь осторожничаете, это, возможно, не столь уж плохая мысль.

Встроенные документы («here» documents)

Строчно-ориентированная форма расстановки кавычек основана на синтаксисе *встроенных документов* (*here-document*) оболочки UNIX. Под строчной ориентированностью имеется в виду, что ограничителями являются не символы, а строки. Начальным ограничителем является текущая строка исходного кода, а конечным – последовательность символов, которую вы задаете. Вслед за << задается последовательность символов, которая должна завершить «цитируемый» текст, и все строки, следующие за текущей, но исключая завершающую, становятся частью строки. Завершающая последовательность может быть идентификатором (словом) или некоторым текстом, заключенным в кавычки. В случае применения кавычек их тип определяет способ обработки текста, как и при обычном использовании кавычек. Идентификатор без кавычек читается так, как если бы был заключен в двойные кавычки. Идентификатор, которому предшествует обратная косая черта, читается так, как если бы был заключен в одинарные кавычки (для совместимости с синтаксисом оболочки). Между << и идентификатором без кавы-

---

1 $LIST_SEPARATOR, если используется модуль English, поставляемый с Perl.

2 Полное описание алгоритма угадывателя не представляет интереса. Скажем лишь, что в его основе лежит сравнение взвешенного среднего всего, что похоже на классы символов (a-z, \w, якорь ^), с тем, что похоже на выражения (переменных и зарезервированных слов).

чек не должно быть пробела, хотя пробельный символ разрешен, если указывается не голый идентификатор, а строка в кавычках. (Если вставить пробел, то он будет считаться за пустой идентификатор, что допустимо, но осуждается, и соответствует первой пустой строке – см. первое `Ура!` **в следующем примере.) Завершающая последовательность должна располагаться на последней строке в чистом виде, без кавычек и лишних пробельных символов по концам.**

```
print <<EOF;      # то же, что в прежнем примере
Цена $Price.
EOF

print <<"EOF";    # то же, что выше, но с явными кавычками
Цена $Price.
EOF

print <<'EOF';    # Одинарные кавычки
All things (e.g. a camel's journey through
A needle's eye) are possible, it's true
But picture how the camel feels, squeezed out
In one long bloody thread, from tail to snout[1]
                              -- C.S. Lewis

EOF

print <<\EOF;     # другой способ указать одинарные кавычки
Сейчас мне сильно пригодились бы $100
EOF

print << x 10;    # вывести следующую строку 10 раз
Верблюды идут! Ура! Ура!

print <<"" x 10; # предпочтительный вариант того же
Верблюды идут! Ура! Ура!

print <<`EOC`;    # выполнить операторы
echo эй, там!
echo смотри туда!
EOC

print <<"верблюд одногорбый", <<"camelid";  # можно собрать в кучу
Я сказал верблюд двугорбый
верблюд одногорбый
Она сказала лама
camelid

funkshun(<<"ЭТО", 23, <<'ТО'); # то, что они в скобках, не имеет значения
Вот строка
или две строки.
ЭТО
```

[1] **Да, это правда, – многое, и даже все – возможно,**
(Верблюд преодолеет узкие врата игольного ушка).
Но каково быть превращенным в кровавое и длинное страдание
От морды до хвоста? – Клайв Стейплз Льюис, «Эпитафии и эпиграммы».

```
Вот еще одна.
TO
```

Не забывайте ставить точку с запятой в конце оператора, поскольку Perl не догадается, что вы не собираетесь делать следующее:

```
print <<'odd'
2345
odd
    + 10000; # выведет 12345
```

Чтобы встроенный документ имел такой же отступ, как в остальном коде, придется вручную удалить пробельные символы в начале каждой строки:

```
(my $quote = <<'QUOTE') =~ s/^\s+//gm;
    The Road goes ever on and on,
    down from the door where it began.1
QUOTE
```

Можно даже заполнить массив строчками встроенного документа, например:

```
my @sauces = <<End_Lines =~ m/(\S.*\S)/g;
    обычный томатный
    томатный со специями
    зеленый чили
    песто
    белое вино
End_Lines
```

Литералы версий Perl

Литерал, начинающийся с буквы v, за которой следует одно или несколько целых чисел, разделенных точками, считается номером версии:

```
use v5.14, # включить строгие ограничения и предупреждения
```

(Раньше такие литералы назывались *v-строками*, но в наши дни их использование для создания строковых значений не приветствуется. Теперь такую форму записи можно использовать только для определения версий.)

Другие литеральные лексемы

Все идентификаторы, начинающиеся и оканчивающиеся удвоенным символом подчеркивания, зарезервированы в Perl для особых синтаксических целей. Двумя такими особыми литералами являются `__LINE__` и `__FILE__`, которые представляют текущие номер строки и имя файла в данной точке вашей программы. Их можно использовать только как отдельные лексемы, они не интерполируются в строки. Аналогично, `__PACKAGE__` представляет собой имя пакета, в который компилируется текущий код. Лексему `__END__` (а также символы Control-D или Control-Z) можно использовать для обозначения логического окончания сценария до фактического конца файла. Любой следующий за ними текст игнорируется, но может быть прочитан с помощью файлового дескриптора DATA.

[1] Из «Властелина колец» Дж. Р.Р. Толкиена. «Дорога вдаль и вдаль ведет, в ее начале – мой порог» (пер. Г. Виноградов). – *Прим. ред.*

Лексема __DATA__ действует аналогично лексеме __END__, но открывает дескриптор файла DATA в пространстве имен текущего пакета, поэтому в файлах, загружаемых посредством require, могут одновременно быть открыты собственные дескрипторы DATA. За дополнительными сведениями обратитесь к описанию DATA в главе 25.

Контекст

Мы уже видели несколько термов, которые могут порождать скалярный контекст. Однако прежде чем продолжить обсуждение термов, следует договориться о значении самого понятия *контекста*.

Скалярный и списочный контексты

Всякая операция[1] в сценарии Perl выполняется в конкретном контексте, и характер ее действия может зависеть от требований этого контекста. Есть два основных контекста: скалярный и списочный. Например, присваивание скалярной переменной или скалярному элементу массива или хеша приводит к вычислению правой части в скалярном контексте:

```
$x        = funkshun(); # скалярный контекст
$x[1]     = funkshun(); # скалярный контекст
$x{"ray"} = funkshun(); # скалярный контекст
```

Однако присваивание массиву или хешу либо срезу того или другого вычисляет правую часть в *списочном контексте*, даже если в срезе выбран только один элемент:

```
@x        = funkshun(); # списочный контекст
@x[1]     = funkshun(); # списочный контекст
@x{"ray"} = funkshun(); # списочный контекст
%x        = funkshun(); # списочный контекст
```

Присваивание списку скаляров тоже обеспечивает списочный контекст в правой части, даже если в списке только один элемент:

```
($x,$y,$z) = funkshun(); # списочный контекст
($x)       = funkshun(); # списочный контекст
```

Правила остаются неизменными при объявлении переменной путем модификации терма с помощью my, state или our, поэтому:

```
my $x   = funkshun(); # скалярный контекст
my @x   = funkshun(); # списочный контекст
my %x   = funkshun(); # списочный контекст
my ($x) = funkshun(); # списочный контекст
```

Не видать вам счастья, пока вы не усвоите различие между скалярным и списочным контекстами, поскольку некоторые операторы (такие как гипотетическая функция funkshun, приведенная нами выше) осознают свой контекст и возвраща-

[1] В данном случае мы свободно пользуемся термином «операция» для обозначения либо оператора, либо терма. Различие между этими двумя понятиями размывается, если говорить о функциях, анализируемых как термы, но выглядящих как унарные операторы.

ют список в контекстах, требующих списка, но скалярное значение в контекстах, требующих скаляра. (Если это верно для какой-либо операции, об этом говорится в ее документации.) На компьютерном жаргоне говорят, что операции *перегружаются* на основе типа возвращаемого значения. Однако это очень простой вид перегрузки, основывающийся только на различии между единственным и множественными значениями и ни на чем ином.

Если некоторые операторы реагируют на контекст, то, очевидно, их среда существования должна каким-то образом этот контекст сообщать. Мы показали, что присваивание способно предоставлять контекст своему правому операнду, но это не должно удивлять, поскольку все операторы предоставляют тот или иной контекст для каждого из своих операндов. Что нас действительно интересует, так это то, *какие* операторы *какой* контекст предоставляют своим операндам. Оказывается, отличить операторы, порождающие списочный контекст, легко, поскольку в их синтаксических описаниях фигурирует слово *LIST*, т.е. *СПИСОК*. Все остальное порождает контекст скалярный. Обычно все вполне очевидно.[1] При необходимости можно установить скалярный контекст для аргумента в середине *LIST* с помощью псевдофункции `scalar`. В Perl нет средства принудительной установки списочного контекста в скалярном контексте, поскольку там, где требуется списочный контекст, он уже обеспечен с помощью *LIST* какой-либо управляющей функции.

Скалярный контекст может быть подразделен на строковый, числовой и нейтральный контексты. В противоположность различению скалярного и списочного контекстов, операции не имеют возможности и необходимости знать, в котором из скалярных контекстов они находятся. Они просто возвращают требуемое скалярное значение и предоставляют Perl переводить числа в строки в строковом контексте и строки в числа в числовом контексте. В некоторых скалярных контекстах не важно, возвращается ли строка, число или ссылка, поэтому преобразование не осуществляется. Это происходит, например, в случае присваивания значения другой переменной. Новая переменная просто получает тот же подтип, что и старое значение.

Логический (булев) контекст

Другим особым произвольным скалярным контекстом является *логический контекст*. Логический контекст представляет собой просто любое место, где вычисляется значение выражения и проверяется его истинность. Когда в этой книге мы говорим «true» или «false», то имеем в виду используемое в Perl техническое определение: скалярное значение является истинным, если оно не равно пустой строке `""` или числу `0` (или его строковому эквиваленту, `"0"`). Ссылка всегда истинна, поскольку представляет адрес, а адрес не может быть нулевым. Неопределенное значение (часто обозначаемое `undef`) всегда ложно, поскольку является `""` или `0`, в зависимости от того, рассматривается ли это значение как строка или как число. (Списочные величины не имеют логического значения, поскольку никогда не создаются в скалярном контексте!)

---

[1] Обратите, однако, внимание на то, что списочный контекст *LIST* может распространяться на вызов подпрограмм, поэтому не всегда очевидно, будет ли данная команда выполняться в скалярном или списочном контексте. Программа может определить свой контекст в подпрограмме с помощью функции `wantarray`.

Поскольку логический контекст является нейтральным контекстом, он не вызывает преобразования скаляров, хотя, конечно, сам скалярный контекст налагается на любой операнд, различающий контекст. А для многих различающих операндов скаляр, создаваемый ими в скалярном контексте, представляет собой некоторое булево значение. Это значит, что операторы, создающие список в списочном контексте, могут быть использованы для проверки на `true`/`false` в логическом контексте. Например, в списочном контексте, производимом оператором `unlink`, имя массива порождает список его значений:

```
unlink @files; # Удалить все файлы, игнорируя ошибки.
```

Но если использовать массив в условном операторе (т.е. в логическом контексте), то массив определяет, что находится в скалярном контексте, и возвращает число элементов в массиве, которое является истинным значением, пока в массиве есть какие-нибудь элементы. Поэтому, если желательно вывести предупреждения для всех файлов, которые не были должным образом удалены, можно написать такой цикл:

```
while (@files) {
    my $file = shift(@files);
    unlink($file) || warn "Невозможно удалить $file: $!
}
```

Здесь `@files` вычисляется в логическом контексте, порожденным оператором `while`, поэтому Perl вычисляет сам массив и проверяет, является ли он «истинным массивом» или «ложным массивом». Истинным массивом он остается, пока в нем содержатся имена файлов, но становится ложным, как только из него выталкивается последнее имя файла. Обратите внимание, что остается справедливым сказанное нами выше. Несмотря на то что массив содержит (и может порождать) списочное значение, мы не вычисляем списочное значение в скалярном контексте. Мы сообщаем массиву, что он скаляр, и спрашиваем, что он сам о себе думает.

Не поддавайтесь соблазну использовать для этих целей конструкцию `defined @files`. Ничего не получится, потому что функция `defined` определяет, является ли скаляр равным `undef`, но массив – это не скаляр. Достаточно простой логической проверки.

Пустой контекст

Еще одним специфическим видом скалярного контекста является *пустой контекст*. В этом контексте не только безразлично, какого типа значение возвращается, но и не требуется возвращать значение. С точки зрения работы функций, он не отличается от обычного скалярного контекста. Но если включен вывод предупреждений, компилятор Perl сообщает о применении выражения без побочных эффектов в таком месте, где не требуется значение, например, в операторе, не возвращающем значение. Например, если в качестве оператора используется строка:

```
"Camel Lot";
```

можно получить предупреждение такого вида:

```
Useless use of a constant in void context in myprog line 123;
[бесполезное использование константы в пустом контексте в myprog, строка 123]
```

Интерполирующий контекст

Выше мы говорили, что в литеральных строках в двойных кавычках производится интерполяция обратной косой черты и переменных, но интерполирующий контекст (часто называемый контекстом двойных кавычек, поскольку выговорить «интерполирующий» невозможно) применяется не только к строкам в двойных кавычках. К другим конструкциям подобного типа относятся обобщенный оператор обратных кавычек qx//, оператор поиска по шаблону m//, оператор подстановки s/// и оператор цитирования регулярного выражения qr//. Оператор подстановки производит интерполяцию в левой части перед поиском по шаблону, а затем интерполяцию в правой части при каждом нахождении соответствия левой части.

Интерполирующий контекст возникает только внутри кавычек или конструкций, работающих как кавычки, поэтому, может быть, не совсем справедливо называть его контекстом в таком же смысле, как скалярный и списочный контексты. (А может быть, и справедливо.)

Списочные значения и массивы

Теперь, когда мы обсудили контексты, можно поговорить о списочных литералах и поведении списочных литералов в контексте. Вы уже знакомы со списочными литералами. Списочные литералы образуются путем разделения отдельных значений запятыми (и заключения списка в круглые скобки, если это требуется для расстановки приоритетов). Поскольку (почти) никогда не вредно добавить пару лишних скобок, синтаксическая диаграмма списочного значения обычно обозначается так:

```
(LIST)
```

Ранее мы говорили, что *LIST* в описании синтаксиса указывает на нечто, предоставляющее списочный контекст для своих аргументов, но голый списочный литерал является некоторым исключением из этого правила, поскольку предоставляет списочный контекст своим аргументам, только когда список в целом находится в списочном контексте. Значениями списочного литерала в списочном контексте являются просто значения аргументов в заданном порядке. В качестве необычной разновидности терма в выражении списочный литерал просто проталкивает ряд временных значений в стек Perl, откуда позже они изымаются оператором, ожидающим получить список. Однако в скалярном контексте списочный литерал ведет себя не так, как *LIST*, поскольку не предоставляет списочный контекст своим значениям. Вместо этого он просто вычисляет каждый из своих аргументов в скалярном контексте и возвращает значение последнего элемента. Это происходит потому, что в действительности он представляет собой закамуфлированный оператор запятой из C, т.е. бинарным оператором, который всегда отбрасывает значение слева и возвращает значение справа. Пользуясь ранее установленным словарем, мы можем сказать, что левая сторона оператора запятой, в сущности, предоставляет пустой контекст. Поскольку оператор запятой обладает левой ассоциативностью, то ряд значений, разделяемых запятыми, приводит в итоге к последнему значению, так как последняя запятая отбрасывает все, что произвели предыдущие запятые. Сравним контексты. Списочное присваивание

```
@stuff = ("один", "два", "три");
```

присваивает все списочное значение массиву @stuff, тогда как скалярное присваивание:

```
$stuff = ("один", "два", "три");
```

присваивает только значение «три» переменной $stuff. Подобно упоминавшемуся выше массиву @files, оператор запятой знает, находится он в скалярном или списочном контексте, и действует соответствующим образом.

Стоит повторить: списочное значение и массив – это разные вещи. Действительная переменная массива знает также свой контекст, и в списочном контексте возвращает свой внутренний список значений как списочный литерал. Но в скалярном контексте она возвращает только размер массива. В следующем примере переменной $stuff присваивается значение 3:

```
@stuff = ("один", "два", "три");
$stuff = @stuff;
```

Если вы ожидали увидеть значение "три", то, вероятно, сделали неверное обобщение, предположив, что Perl использует правило оператора запятой, чтобы отбросить все, кроме одного, временные значения, которые помещены @stuff в стек. Но дело обстоит не так. Массив @stuff не помещал все свои значения в стек. В действительности, он не помещал в стек ни одного своего значения. Он поместил лишь одну величину, размер массива, поскольку *знал*, что находится в скалярном контексте. Ни один терм или оператор в скалярном контексте не помещает в стек список. Вместо этого он помещает в стек один скаляр (каким бы непривычным это ни показалось), который едва ли окажется последним значением списка, которое он возвратил *бы* в списочном контексте, потому что последнее значение вряд ли окажется самым полезным значением в скалярном контексте. Уловили? (Если нет, лучше перечитайте этот абзац еще раз, поскольку это важно.)

Теперь вернемся к настоящим *LIST* – тем, которые образуют списочный контекст. До сих пор мы представляли дело так, будто списочные литералы являются просто списками литералов. Но подобно тому, как строковый литерал может интерполировать другие подстроки, списочный литерал может интерполировать другие подсписки. В списке может находиться любое выражение, возвращающее значения. Используемые при этом значения могут быть скалярными или списочными, но все они становятся частью нового списочного значения, потому что *LIST* производит автоматическую интерполяцию подсписков. Это означает, что при вычислении *LIST* каждый элемент списка вычисляется в списочном контексте, а полученное списочное значение интерполируется в *LIST*, как если бы каждый отдельный элемент был членом *LIST*. Поэтому массивы утрачивают свою индивидуальность в *LIST*.[1] Список

```
(@stuff,@nonsense,funkshun())
```

содержит элементы @stuff, за которыми следуют элементы @nonsense, а за ними следуют те значения, которые подпрограмма &funkshun сочтет нужным вернуть в списочном контексте. Обратите внимание, что любая позиция из списка выше может интерполироваться в пустой список, и в этом случае все происходит так,

---

[1] Некоторым кажется, что данное обстоятельство представляет собой проблему, но это не так. Всегда можно интерполировать ссылку на массив, если вы не хотите, чтобы он утратил свою индивидуальность. См. главу 8.

как если бы в этом месте не интерполировался никакой массив или вызов функции. Собственно пустой список представляется литералом (). Как и для пустого массива, который интерполируется как пустой список и потому фактически игнорируется, интерполяция пустого списка в другой список не оказывает никакого эффекта. Поэтому ((),(),()) эквивалентно ().

Следствием этого правила является возможность поместить необязательную запятую в конце любого списочного значения. Это упростит добавление элементов в конец списка в будущем:

```
@releases = (
    "альфа",
    "бета"
    "гамма",
);
```

Можно вообще отказаться от запятых: другим способом задания списка литералов является упомянутый выше синтаксис qw (quote words, цитирование слов). Эта конструкция эквивалентна расщеплению строки в одинарных кавычках по пробельным символам. Например:

```
@froots = qw(
    яблоко    банан      карамболь
    кокос     гуава      кумкват
    мандарин  нектарин   персик
    груша     хурма      слива
);
```

(Обратите внимание, что эти круглые скобки ведут себя не как обычно, а как кавычки. С таким же успехом это могли быть угловые или фигурные скобки или косая черта, прямая или обратная. Но круглые скобки красивее.)

Со списочным значением можно использовать индексы, как в обычном массиве. Во избежание неоднозначности следует заключать список в круглые скобки (настоящие). Хотя таким способом часто из списка извлекаются отдельные значения, на самом деле это срез списка, поэтому синтаксис такой:

```
(LIST)[LIST]
```

Примеры:

```
# stat возвращает списочное значение
$modification_time = (stat($file))[9];

# ЗДЕСЬ СИНТАКСИЧЕСКАЯ ОШИБКА.
$modification_time = stat($file)[9], # ОЙ, СКОБКИ ЗАБЫЛИ

# Найти шестнадцатеричную цифру.
$hexdigit = ('a'.'b'.'c','d','e'.'f')[$digit-10];

# "Обратный оператор запятой"
return (pop(@foo),pop(@foo))[0];

# Получить несколько значений как срез.
($day, $month, $year) = (localtime)[3,4,5];
```

Списочное присваивание

Присваивание списку можно осуществлять, только если допустимо присваивание для каждого элемента списка:

```
($a, $b, $c) = (1, 2, 3);

($map{red}, $map{green}, $map{blue}) = (0xff0000, 0x00ff00, 0x0000ff);
```

Можно осуществлять присваивание неопределенным элементам (undef) в списке. Это удобно, если нужно отбросить некоторые значения, возвращаемые функцией:

```
($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

Это можно делать даже в объявлениях my:

```
my ($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

Последним элементом списка может быть массив или хеш:

```
($a, $b, @rest) = split;
my ($a, $b, %rest) = @arg_list;
```

Поместить массив или хеш можно в любом месте списка, которому производится присваивание, но первый массив или хеш, встретившийся в списке, поглотит все оставшиеся значения, и все последующие элементы списка получат неопределенные значения. Это можно использовать в local или my, когда нужно, чтобы инициализированные массивы остались пустыми.

Можно даже осуществлять присваивание пустому списку:

```
() = funkshun();
```

Это приводит к вызову функции в списочном контексте с отбрасыванием возвращаемых значений. Вызови мы такую функцию без присваивания, вызов произошел бы в пустом контексте, который является разновидностью скалярного контекста, и при этом функция могла бы повести себя совершенно иначе. Присваивание списку в скалярном контексте возвращает число элементов, образованных выражением в *правой* части присваивания:

```
$x = ( ($a, $b) = (7,7,7) );# присваивает $x 3, а не 2
$x = ( ($a, $b) = funk() ); # присваивает $x количество значений, возвращаемых funk()
$x = ( () = funk() );       # тоже присваивает $x число значений, возвращаемых funk()
```

Это удобно в случае присваивания списку в логическом контексте, поскольку большинство списочных функций возвращает по завершении пустой список, который при присваивании порождает 0, интерпретируемый как false. Вот как можно использовать это в цикле while:

```
while (($login, $password) = getpwent) {
    if (crypt($login, $password) eq $password) {
        print "$login имеет ненадежный пароль!\n";
    }
}
```

Размер массива

Число элементов в массиве @days можно определить, вычислив @days в скалярном контексте, например:

```
@days + 0;     # неявная установка скалярного контекста для @days
scalar(@days) # явная установка скалярного контекста для @days
```

Обратите внимание, что это действует только для массивов. В общем случае для списочных значений это не работает. Как уже говорилось, список с разделителем-запятой, вычисляемый в скалярном контексте, возвращает последнее значение, как оператор запятой в C. Но поскольку в Perl практически никогда не требуется знать размер списка, из-за этого не возникает проблем.

Со скалярным вычислением @days тесно связана переменная $#days. Она возвращает индекс последнего элемента массива, на единицу меньший размера массива, поскольку (обычно) имеется нулевой элемент. Присваивание переменной $#days изменяет размер массива. Укорачивание массива таким способом уничтожает лишние значения. Можно добиться некоторого увеличения эффективности программы, заранее увеличивая размер массива, который должен разрастись. (Расширить массив можно и присваиванием значения элементу, находящемуся за его пределами.) Можно произвести полное усечение массива, присвоив ему пустой список (). Следующие строки эквивалентны:

```
@whatever = ();
$#whatever = -1;
```

Всегда выполняется и следующее равенство:

```
scalar(@whatever) == $#whatever + 1;
```

Усечение массива не высвобождает занимаемую им память. Чтобы вернуть память в пул памяти процесса, нужно использовать undef(@whatever) или же выходом за пределы области видимости. Вернуть память в пул памяти операционной системы, вероятно, не удастся, поскольку мало какие операционные системы поддерживают это.

Хеши

Как было сказано ранее, хеш является просто необычным видом массива, поиск значений в котором осуществляется с помощью ключевых строк, а не чисел. Хеш ассоциирует ключи со значениями, поэтому хеши часто называют *ассоциативными массивами* (те, кому не лень набирать такое длинное название).

На практике в Perl нет такой вещи, как хеш-литерал, но если присвоить хешу обычный список, то каждая пара значений в списке образует одну пару ключ/значение:

```
my %map = ("red",0xff0000,"green",0x00ff00,"blue",0x0000ff);
```

Такой же результат получается после выполнения операторов:

```
my %map         # неинициализированный хеш пуст
$map{red}   = 0xff0000;
$map{green} = 0x00ff00;
$map{blue}  = 0x0000ff;
```

Часто код легче читать, если использовать оператор => в парах ключ/значение. Оператор => служит просто синонимом запятой, но более нагляден и заключает в кавычки голые идентификаторы в своей левой части (как идентификаторы в фи-

гурных скобках выше), что делает его удобным для нескольких видов операций, включая инициализацию переменных типа хеш:

```
my %map = (
    red   => 0xff000C,
    green => 0x00ff00,
    blue  => 0x0000ff,
);
```

или инициализацию ссылок на анонимные хеши, используемые как записи:

```
my $rec = {
    NAME  => "John Smith".
    RANK  => "Captain"
    SERNO => "951413",
};
```

или применение именованных параметров для вызова сложных функций:

```
my $field = radio_group(
    NAME      => "животные",
    VALUES    => ["верблюд", "лама", "баран", "волк"],
    DEFAULT   => "верблюд",
    LINEBREAK => "true",
    LABELS    => \%animal_names,
)
```

Но мы снова забегаем вперед. Вернемся к хешам.

Переменную типа хеш (%hash) можно использовать в списочном контексте, и в этом случае она интерполирует все свои пары ключ/значение в список. То, что хеш инициализировался в определенном порядке, не означает, что его значения будут возвращаться в том же порядке. В реализации хешей применяются хеш-таблицы (для быстрого поиска), в результате чего порядок, в котором элементы хранятся, зависит от внутренней хеш-функции, используемой для расчета позиции в хеш-таблице, а не чего-либо достойного внимания. Поэтому записи возвращаются в порядке, кажущемся случайным. (Разумеется, порядок элементов каждой пары ключ/значение сохраняется.) Примеры способов упорядочения вывода можно найти в описании функции keys в главе 27.

При вычислении хеш-переменной в скалярном контексте значение true возвращается, если в хеше есть хотя бы одна пара ключ/значение. Если пары ключ/значение есть, возвращаемое значение является строкой, состоящей из числа использованных блоков и числа выделенных блоков, разделенных косой чертой. В основном это полезно, только чтобы обнаружить, что встроенный алгоритм хеширования Perl плохо работает на ваших данных. Например, если вы поместили в хеш 10 000 значений, и при вычислении %HASH в скалярном контексте получили "1/8", это означает, что лишь один из восьми блоков используется для хранения. Вероятно, в этом одном блоке и содержатся все 10 000 ваших элементов. Скорее всего, такого не произойдет.

Чтобы определить количество ключей в хеше, вызовите функцию keys в скалярном контексте:

```
scalar(keys(%HASH)).
```

Многомерный хеш можно эмулировать путем задания в фигурных скобках нескольких ключей, разделенных запятыми. Перечисленные ключи конкатениру-

ются, при этом в качестве разделителя выступает содержимое переменной `$;` (`$SUBSCRIPT_SEPARATOR`), по умолчанию равное `chr(28)`. Получившаяся строка используется в качестве фактического ключа хеша. Две следующие строки делают одно и то же:

```
$people{ $state, $county } = $census_results;
$people{ join $; => $state, $county } = $census_results;
```

Эта функция первоначально была реализована для поддержки *a2p*, транслятора из awk в Perl. В нынешние времена вы можете просто использовать настоящий (скажем, более настоящий) многомерный массив, как это описывается в главе 9. Но в одном случае старый стиль все еще удобен: для хешей, связанных с внешними файлами, не поддерживающими многомерные ключи, такие как файлы DBM.

Не путайте эмуляцию многомерных хешей со срезами. Одно представляет скалярное значение, а другое – списочное:

```
$hash{ $x, $y, $z } # одиночное значение
@hash{ $x, $y, $z } # срез из трех значений
```

Таблицы имен и дескрипторы файлов

Особый тип Perl под названием *typeglob* предназначается для хранения целой записи таблицы имен. (Запись таблицы имен `*foo` содержит значения `$foo`, `@foo`, `%foo`, `&foo` и нескольких интерпретаций старого доброго `foo`.) Разыменовывающим префиксом для типа typeglob является символ `*`, поскольку он представляет все типы.

Одним из применений типа typeglob (или ссылок на него) является передача и запоминание дескрипторов файлов – это было особенно актуально до появления в Perl ссылок на дескрипторы файлов. Чтобы запомнить голословный дескриптор файла, поступите так:

```
$fh = *STDOUT;
```

или используйте реальную ссылку:

```
$fh = \*STDOUT;
```

или можно обратиться к разделу таблицы, где хранятся дескрипторы файлов:

```
$fh = *STDOUT{IO}
```

Раньше это было предпочтительным способом создания локального дескриптора файла. Например:

```
sub newopen {
    my $path = shift;
    local *FH,           # ни my(), ни our()
    open(FH, '< , $path) || return undef;
    return *FH;          # не \*FH!
}
$fh = newopen("/etc/passwd")
```

Однако в настоящее время практически всегда лучше позволить Perl самостоятельно выбрать имя для дескриптора файла:

```
sub newopen {
    my $path = shift;
    open(my $fh, '<', $path) || return undef;
```

```
    return $fh;
}
$fh = newopen("/etc/passwd");
```

Сегодня основным применением typeglob оказывается создание псевдонима для записи таблицы имен. Псевдоним можно рассматривать как прозвище. Если сказать:

```
*foo = *bar;
```

то все с именем foo становится синонимом для соответствующего объекта с именем bar. Можно создать псевдоним для отдельной переменной из таблицы имен, присвоив ссылку:

```
*foo = \$bar;
```

При этом $foo становится псевдонимом для $bar, но @foo не становится псевдонимом для @bar, как и %foo для %bar. Все это оказывает воздействие только на глобальные (пакетные) переменные; доступ к лексическим переменным не осуществляется через записи таблицы имен. Такое создание псевдонимов для глобальных переменных может показаться делом совершенно ненужным, однако на этой функции построен весь механизм экспорта/импорта модулей, поскольку нигде не сказано, что символ, для которого создается псевдоним, должен находиться в том же пространстве имен. Конструкция

```
local *Here::blue = \$There::green;
```

временно делает $Here::blue псевдонимом для $There::green, но не делает @Here::blue псевдонимом для @There::green или %Here::blue псевдонимом для %There::green. К счастью, все эти сложные манипуляции с typeglob скрыты от ваших глаз. Дополнительное обсуждение typeglob и вопросов импорта можно найти в разделах «Ссылки на дескрипторы» и «Ссылки на таблицы имен» главы 8, в разделе «Таблицы имен» главы 10, а также в главе 11.

Операторы ввода

Есть несколько операторов ввода, которые мы опишем здесь, поскольку они анализируются как термы. Иногда их называют псевдолитералами, поскольку во многих отношениях они ведут себя как строки в кавычках. (Операторы вывода типа print анализируются как списочные операторы и обсуждаются в главе 27.)

Оператор ввода команд (обратные кавычки)

Прежде всего, в нашем распоряжении имеется оператор ввода команд, известный также как оператор «обратные кавычки» (backticks), поскольку он выглядит так:

```
$info = `perldoc $module`;
```

Строка, заключенная в обратные кавычки (или, говоря «техническим» языком, в грависы), сначала претерпевает интерполяцию переменных, как строка в двойных кавычках. Затем результат интерпретируется системой как командная строка, а выдача команды становится значением псевдолитерала. (Это делается на манер аналогичного оператора оболочек UNIX.) В скалярном контексте возвращается одна строка, содержащая выдачу. В списочном контексте возвращается

список значений, по одному на каждую строку выдачи. (С помощью переменной $/ можно назначить иной символ окончания строки.)

Команда выполняется при всяком вычислении псевдолитерала. Числовой код завершения команды сохраняется в переменной $? (см. в главе 25 интерпретацию $?, известную также, как $CHILD_ERROR). В отличие от версии этой команды в *csh*, в возвращаемых данных не производится трансляция: перевод строки остается переводом строки. В отличие от всех оболочек, одинарные кавычки в Perl не защищают имена переменных в команде от интерпретации. Чтобы передать символ $ оболочке, нужно защитить этот символ обратной косой чертой. Переменная $module в вышеприведенном примере *finger* интерполируется интерпретатором Perl, а не оболочкой. (Поскольку команда подвергается обработке оболочкой, возникают вопросы защиты данных, рассматриваемые в главе 20.)

Обобщенной формой «обратных кавычек» является qx// (от «quoted execution» – «выполнение в кавычках»), но оператор действует в точности так же, как обычные обратные кавычки. Просто у вас появляется возможность выбрать собственные символы кавычек. Как и в случае аналогичных псевдофункций цитирования, если в качестве разделителя выбирается одинарная кавычка, команда не подвергается «интерполяции двойных кавычек»:

```
$perl_info  = qx(ps $$);  # это $$ в Perl
$shell_info = qx'ps $$'   # это $$ в оболочке
```

Оператор ввода строки (угловых скобок)

Из операторов ввода наиболее интенсивно используется оператор ввода строки, который называют также оператором угловых скобок и функцией readline (поскольку внутренне он вызывает именно ее). Вычисление дескриптора файла в угловых скобках (например, STDIN) дает следующую строку файла, ассоциированного с дескриптором. (Символ перевода строки включается в состав строки, поэтому, согласно критериям истинности Perl, вновь прочитанная строка всегда является истинной, пока не достигнут конец файла – в этот момент возвращается неопределенное значение, которое, к счастью, является ложным.) Обычно входное значение присваивается переменной, но в одном случае происходит автоматическое присваивание. Значение автоматически присваивается специальной переменной $_ – тогда и только тогда, когда в условии цикла while нет ничего, кроме оператора ввода строки. При этом проверяется, определено ли присвоенное значение. (Эта конструкция может показаться странной, но вы будете часто ее использовать, поэтому стоит ее изучить.) Как бы там ни было, следующие строки эквивалентны:

```
while (defined($_ = <STDIN>)) { print $_ } # самый длинный путь
while ($_ = <STDIN>) { print }             # явно относительно $_
while (<STDIN>) { print }                  # короткий способ
for (;<STDIN>;) { print }                  # цикл while в камуфляже
print $_ while defined($_ = <STDIN>);      # длинный модификатор оператора
print while $_ = <STDIN>,                  # явно относительно $_
print while <STDIN>;                       # короткий модификатор оператора
```

Помните, что для этого особого применения требуется цикл while. Когда оператор ввода используется в любом другом месте, результат должен присваиваться явно, если требуется сохранить его значение:

```
while (<FH1> && <FH2>) {     } # НЕВЕРНО: отбрасывает оба ввода
if (<STDIN>) { print }         # НЕВЕРНО: выводит прежнее значение $_
if ($_ = <STDIN>) { print }    # не самое оптимальное: нет проверки определенности
if (defined($_ = <STDIN>)) { print }     # самое лучшее
```

При неявном присваивании $_ в цикле, использующем $_, это имя указывает на глобальную переменную, а не локальную для цикла while. Существующее значение $_ можно защитить так:

```
while (local $_ = <STDIN>) { print } # использовать локальную $_
```

или так:

```
while (my $_ = <STDIN>) { print }    # новая, лексическая $_
```

Любое предшествующее значение восстанавливается по завершении цикла. Однако если не использовать объявление my или state, переменная $_ остается глобальной, поэтому функции, вызываемые изнутри цикла, могут обращаться к ней явным или неявным образом. Этого также можно избежать, объявив лексическую переменную:

```
while (my $line = <STDIN>) { print $line } # теперь локальная
```

(Оба эти цикла while все же неявно проверяют, является ли определенным результат присваивания, поскольку my, state и local не влияют на то, как анализатор воспринимает присваивание.) Дескрипторы файлов STDIN, STDOUT и STDERR являются предопределенными и заранее открытыми. Дополнительные дескрипторы файлов могут создаваться с помощью функций open или sysopen. Эти функции описываются в главе 27.

В приведенных выше циклах while оператор ввода строки вычислялся в скалярном контексте, поэтому возвращал каждую строку отдельно. Однако если использовать этот оператор в списочном контексте, возвращается список, состоящий из всех оставшихся строк, по одной строке на каждый элемент. Так можно легко создать *огромное* пространство данных, поэтому применяйте эту функцию с осторожностью:

```
$one_line  = <MYFILE>; # Получить первую строку.
@all_lines = <MYFILE>; # Получить все оставшиеся строки
```

Со списочной формой оператора ввода не связаны никакие чудеса while, поскольку условие цикла while всегда предоставляет скалярный контекст (как и всякий условный оператор).

Использование пустого дескриптора файла в операторе угловых скобок представляет собой особый случай: оно эмулирует режим командной строки таких типичных программ-фильтров UNIX, как *sed* и *awk*. При чтении строк из <> вы чудесным образом получаете все строки из файлов, перечисленных в командной строке. Если никакие файлы не указаны, вы получаете строки из стандартного потока ввода, поэтому вашу программу легко внедрить в конвейер процессов.

Вот как это работает: при первом вычислении <> проверяется массив @ARGV, и если он является пустым, то $ARGV[0] устанавливается равным "-", что при открытии дает стандартное устройство ввода. После этого массив @ARGV обрабатывается как список имен файлов. Если развернуть цикл

```
while (<>) {
    ...                  # код для каждой строки
}
```

он будет эквивалентен такому псевдокоду в стиле Perl:

```
@ARGV = ('-') unless @ARGV; # STDIN, если список пуст, и только тогда
while (@ARGV) {
    $ARGV = shift @ARGV;      # каждый раз укорачивать @ARGV
    if (!open(ARGV, '<', $ARGV)) {
        warn "Невозможно открыть $ARGV: $!\n";
        next;
    }
    while (<ARGV>) {
        ...                   # код для каждой строки
    }
}
```

за исключением того, что первый цикл не так утомительно писать, и он действительно работает. Он действительно сдвигает массив @ARGV и помещает текущее имя файла в глобальную переменную $ARGV. При этом также используется особый дескриптор файла ARGV, т.е. <> служит просто синонимом более явной записи <ARGV>, являющейся волшебным дескриптором файла. (Псевдокод, приведенный выше, не работает, поскольку не считает <ARGV> за волшебный дескриптор.)

Можно модифицировать @ARGV до первого обращения к <>, если в итоге этот массив будет содержать список тех имен файлов, которые действительно вам нужны. Поскольку Perl использует здесь свою обычную функцию open, то имя файла "-", если таковое встретится, считается стандартным потоком ввода, и вам автоматически становятся доступны более экзотические возможности open (такие как открытие «файла» с именем "gzip -dc < file.gz |"). Нумерация строк ($.) продолжается так, как если бы ввод удачно представлял собой один большой файл. (Однако посмотрите в примере для eof в главе 27, как обнулять номера строк для каждого входного файла.)

Если требуется поместить в @ARGV собственный список файлов, то это можно сделать так:

```
# читать файл README, если не заданы аргументы
@ARGV = ("README") unless @ARGV;
```

При необходимости передать в сценарий ключи командной строки можно сделать это с помощью одного из модулей Getopt::* или выполнить сначала цикл такого типа:

```
while (@ARGV and $ARGV[0] =~ /^-/) {
    $_ = shift;
    last if /^--$/;
    if (/^-D(.*)/) { $debug = $1 }
    if (/^-v/)     { $verbose++ }
    ...            # прочие ключи
}
while (<>) {
    ...            # код обработки каждой строки
}
```

Оператор <> вернет false только один раз. Если после этого его вызвать снова, оператор будет считать, что обрабатывается еще один список @ARGV, и, если значения @ARGV не присвоены, будет читать данные из STDIN.

Если строка в угловых скобках является скалярной переменной (например, <$foo>), то эта переменная содержит *косвенный дескриптор* файла, являющийся либо именем дескриптора файла, из которого нужно брать входные данные, либо ссылкой на такой дескриптор. Например:

```
$fh = \*STDIN;
$line = <$fh>;
```

или:

```
open($fh, '<', "<data.txt");
$line = <$fh>;
```

Поиск файлов по маске

Вас может заинтересовать, что произойдет, если внутрь оператора ввода строки поместить что-нибудь более оригинальное. Произойдет вот что: трансформация в другой оператор. Если строка в угловых скобках не является именем дескриптора файла или скалярной переменной (даже если она просто содержит лишние пробелы), она интерпретируется как маска имени файла, в соответствии с которой должен осуществляться поиск («globbing»).[1] Совпадения с шаблоном отыскиваются в файлах текущего каталога (или каталога, заданного как часть шаблона), и оператор возвращает найденные имена файлов. Как и в случае оператора ввода строки, имена возвращаются по одному в скалярном контексте, или все сразу в списочном контексте. Обычно встречается последний вариант, и нередко можно видеть такие операторы:

```
@files = <*.xml>;
```

Как и для других типов псевдолитералов, сначала выполняется один уровень интерполяции переменных, но для этого нельзя написать <$foo>, поскольку, как сказано выше, это будет воспринято как косвенный дескриптор файла. В прежних версиях Perl программисты вызывали принудительную интерпретацию строки как маски при помощи фигурных скобок: <${foo}>. В настоящее время предпочитают прямой вызов внутренней функции, как в glob($foo), что, вероятно, правильно. Поэтому вместо угловых скобок вы пишете:

```
@files = glob("*.xml")
```

если с презрением относитесь к перегрузке оператора угловых скобок для этих целей. Имеете на это право.

Применяете вы функцию glob или придерживаетесь старой формы с угловыми скобками, в любом случае оператор поиска файла по маске, подобно оператору ввода строки, осуществляет магию цикла while, присваивая результат переменной $_. (Это изначально было главным обоснованием перегрузки оператора угловых скобок.) Например, чтобы изменить права доступа ко всем файлам исходных текстов на C, можно сказать:

---

[1] Fileglob (маска имени файла) не имеет никакого отношения к ранее упомянутым typeglob (шаблон имени символа), за исключением того, что в обоих символ * выступает в качестве группового. У символа *, используемого с этой целью, есть прозвище «glob». С помощью typeglob осуществляется поиск одноименных символов в таблице имен. С помощью fileglob осуществляется поиск имен файлов в каталоге по маске, как это происходит во многих оболочках.

```
while (glob "*.c") {
    chmod 0644, $_;
}
```

что эквивалентно:

```
while (<*.c>) {
    chmod 0644, $_;
}
```

Функция glob в более старых версиях Perl (и еще более старых версиях UNIX) была реализована как команда оболочки, что приводило к относительной дороговизне ее выполнения, а что еще хуже, она по-разному работала в различных системах. Сейчас она является встроенной, а потому более надежной и значительно более быстрой.

Конечно, самый короткий способ выполнения приведенной выше функции chmod и, можно утверждать, самый удобочитаемый, – это использование поиска по маске как списочного оператора:

```
chmod 0644, <*.c>,
```

Оператор поиска по маске вычисляет свой (внедренный) операнд только при создании нового списка. Все значения должны быть прочитаны, прежде чем оператор начнет работу снова. В списочном контексте это не важно, поскольку в любом случае все значения будут автоматически прочитаны. Но в скалярном контексте оператор при каждом вызове возвращает очередное значение или false, если значения кончились. Опять же, false возвращается только один раз. Поэтому, рассчитывая, что поиск по шаблону вернет единственное значение, гораздо лучше сказать:

```
($file) = <blurch*>; # списочный контекст
```

чем:

```
$file = <blurch*>; # скалярный контекст
```

поскольку первая конструкция возвращает все найденные имена и сбрасывает оператор поиска, в то время как вторая поочередно возвращает то имена файлов, то false.

Пытаясь осуществить интерполяцию переменных, явно следует предпочесть оператор glob, поскольку старые обозначения могут быть спутаны с обозначениями косвенных дескрипторов файлов. Здесь становится очевидным, что граница между термами и операторами несколько расплывчата:

```
@files = <$dir/*.[ch]>;          # Действует, но лучше избегать.
@files = glob("$dir/*.[ch]")     # Вызов glob как функции.
@files = glob $some_pattern;     # Вызов glob как оператора
```

В последнем примере мы опустили скобки, чтобы показать, что glob можно применять не только как функцию (терм), но и как *унарный оператор*; т.е. префиксный оператор, принимающий один аргумент. Оператор glob служит примером *именованного унарного оператора*, одного из типов операторов, о которых мы поговорим в следующей главе. Позже мы поговорим еще об операторах поиска по шаблону, которые тоже анализируются как термы, но ведут себя как операторы.

3

Унарные и бинарные операторы

В предыдущей главе речь шла о различных типах термов, которые можно использовать в выражениях, но, по правде говоря, одинокие термы скучноваты. Термы, в большинстве своем, общественные животные. Они любят вступать в связь друг с другом. Типичный молодой терм чувствует сильную потребность в сопричастности, и ему хочется различными способами оказывать влияние на другие термы, однако в обществе существует много видов взаимодействия и много различных уровней приверженности. В Perl эти взаимоотношения выражаются с помощью операторов.

Должна же социология приносить какую-то пользу.

С точки зрения математики, операторы являются обычными функциями с особым синтаксисом. С точки зрения лингвистики – это просто неправильные глаголы. Но любой лингвист обязательно заметит, что неправильные глаголы как раз чаще всего используются в языке. И это важно с точки зрения теории информации, поскольку изложение с применением неправильных глаголов как для говорящего, так и для слушателя является, как правило, более кратким и более эффективным.

Говоря будничным языком, работать с операторами удобно.

Операторы многочисленны, и классифицировать их можно по *арности* (количеству мест для операндов), *приоритету* (степени рвения, с которой они стараются отобрать эти операнды у окружающих операторов) и *ассоциативности* (предпочитаемому порядку работы – слева направо или справа налево – при сочетании с операторами, имеющими такой же приоритет).

По арности различают три вида операторов Perl: *унарные*, *бинарные* и *тернарные*. Унарные операторы всегда являются префиксными (за исключением постфиксных операторов инкрементирования и декрементирования).[1] Все остальные операторы – инфиксные, если не считать списочные операторы, которые могут быть префиксом для любого числа аргументов. Однако большинство воспринимает

---

[1] Хотя различные кавычки и скобки можно рассматривать как «ци́ркумфиксные» операторы, определяющие границы термов.

списочные операторы как обычные функции, вокруг аргументов которых допускается позабыть поставить скобки. Вот несколько примеров:

```
! $x                 # унарный оператор
$x * $y              # бинарный оператор
$x ? $y : $z         # тернарный оператор
print $x, $y, $z     # списковый оператор
```

Приоритет оператора определяет прочность связывания его аргументов. Операторы с более высоким приоритетом захватывают близлежащие аргументы ранее, чем операторы с более низким приоритетом. Классический пример на эту тему пришел к нам прямиком из элементарной математики, в которой умножение имеет более высокий приоритет, чем сложение:

```
2 + 3 * 4       # дает в результате 14, а не 20
```

Порядок выполнения двух операторов с одинаковым приоритетом зависит от их ассоциативности. Эти правила тоже до некоторой степени следуют соглашениям, принятым в математике:

```
2 * 3 * 4       # означает (2 * 3) * 4, левая ассоциативность
2 ** 3 ** 4     # означает 2 ** (3 ** 4), правая ассоциативность
2 != 3 != 4     # недопустимо, не ассоциативный оператор
```

В табл. 3.1 перечислены ассоциативность и арность операторов Perl в порядке убывания приоритета.

Таблица 3.1. Приоритеты операторов

Ассоциативность	Арность	Класс приоритета
Нет	0	Термы и списочные операторы (влево)
Слева	2	`->`
Нет	1	`++ --`
Справа	2	`**`
Справа	1	`! ~ \` и унарные `+` и `-`
Слева	2	`=~ !~`
Слева	2	`* / % x`
Слева	2	`+ -`
Слева	2	`<< >>`
Справа	0, 1	Именованные унарные операторы
Нет	2	`< > <= >= lt gt le ge`
Нет	2	`== != <=> eq ne cmp ~~`
Слева	2	`&`
Слева	2	`| ^`
Слева	2	`&&`
Слева	2	`|| //`
Нет	2	`.. ...`
Справа	3	`?:`

Ассоциативность	Арность	Класс приоритета
Справа	2	`= += -= *=` и т.д.
Слева	2	`, =>`
Справа	0+	Списочные операторы (вправо)
Справа	1	`not`
Слева	2	`and`
Слева	2	`or xor`

Может показаться, что уровней старшинства слишком много, чтобы их можно было запомнить. И это правда. К счастью, есть два облегчающих обстоятельства. Во-первых, уровни приоритетов определены так, что обычно соответствуют интуитивным представлениям, если у вас все в порядке с головой. А во-вторых, те, у кого нервы пошаливают, всегда могут поставить пару лишних скобок, чтобы избавиться от мучительного беспокойства.

Еще один полезный совет: операторы, заимствованные из C, сохраняют по отношению друг к другу принятые там приоритеты, даже когда в C для них установлено довольно странное старшинство. (Это облегчает изучение Perl тем, кто знает C и C++. И, может быть, даже Java.)

В последующих разделах будет рассказано об этих операторах в порядке старшинства. За очень редкими исключениями, все они оперируют только скалярными величинами, а не списками. По необходимости мы будем указывать на исключения из этого правила.

Хотя ссылки и являются скалярными величинами, применение большинства этих операторов к ссылкам не имеет особого смысла, поскольку числовое значение ссылки полезно лишь для внутренних механизмов Perl. Однако если ссылка указывает на объект класса, допускающего перегрузку, то к такому объекту эти операторы можно применять, а если в классе определена перегрузка некоторого оператора, то она определяет действие над объектом при применении к нему данного оператора. Таким способом, например, в Perl реализованы комплексные числа. Более подробно перегрузка рассматривается в главе 13.

Термы и списочные операторы (влево)

Любой *терм* имеет в Perl высший приоритет. В число термов входят переменные, операторы кавычек и подобные им, большинство выражений в круглых, квадратных или фигурных скобках и любые функции, аргументы которых заключены в скобки. Если разобраться, то такая вещь, как функция, не существует, есть только списочные и унарные операторы, ведущие себя как функции, поскольку их аргументы заключены в круглые скобки. Тем не менее, мы назвали главу 27 «Функции».

Теперь слушайте внимательно. Есть пара очень важных правил, существенно упрощающих дело, но приводящих иногда к неожиданным результатам, если не проявлять осмотрительность. Если следующей за списочным оператором (например, `print`) или любым именованным унарным оператором (например, `chdir`) лексемой оказывается левая круглая скобка (не считая пробельных символов), то

оператор и аргументы в скобках получают высший приоритет, как если бы это был обычный вызов функции. Правило такое: если это *похоже* на вызов функции, то *это и есть* вызов функции. Можно сделать так, чтобы он не выглядел как функция, поместив перед скобками унарный плюс, который, семантически говоря, не делает абсолютно ничего – даже не преобразует аргумент в число.

Например, поскольку || имеет более низкий приоритет, чем chdir, мы получим:

```
chdir $foo    || die;       # (chdir $foo) || die
chdir($foo)   || die;       # (chdir $foo) || die
chdir ($foo)  || die;       # (chdir $foo) || die
chdir +($foo) || die;       # (chdir $foo) || die
```

но * имеет более высокий приоритет, чем chdir, поэтому:

```
chdir $foo * 20;            # chdir ($foo * 20)
chdir($foo) * 20;           # (chdir $foo) * 20
chdir ($foo) * 20;          # (chdir $foo) * 20
chdir +($foo) * 20;         # chdir ($foo * 20)
```

Аналогично для любого числового оператора, который является именованным унарным оператором, например оператора rand:

```
rand 10 * 20;               # rand (10 * 20)
rand(10) * 20;              # (rand 10) * 20
rand (10) * 20;             # (rand 10) * 20
rand +(10) * 20;            # rand (10 * 20)
```

В отсутствие круглых скобок приоритет списочных операторов, таких как print, sort или chmod, оказывается либо очень высоким, либо очень низким, в зависимости от того, на какую сторону оператора смотреть – левую или правую. (Это и есть «влево», выведенное в заголовок этого раздела.) Например, в таком коде:

```
my @ary = (1, 3, sort 4, 2);
print @ary;                 # выведет 1324
```

запятые справа от sort вычисляются перед вызовом sort, но запятые слева вычисляются после него. Иными словами, списочный оператор стремится проглотить все аргументы, которые следуют после него, а затем вести себя как простой терм в предшествующем выражении. Но со скобками все же нужно обращаться осторожно:

```
# Эти операторы выполняют exit раньше, чем print:
print($foo, exit)           # Очевидно, это не то, что нужно.
print $foo, exit;           # Как и это.

# Эти операторы выполняют print раньше, чем exit:
(print $foo), exit;         # Это то, что нужно.
print($foo), exit;          # И это
```

Чаще всего обжигаются на использовании скобок для группировки математических аргументов, забывая при этом, что скобки применяются также для группировки аргументов функций:

```
print ($foo & 255) + 1, "\n";       # выведет ($foo & 255)
```

Это работает, скорее всего, не так, как вы рассчитывали.[1] К счастью, ошибки такого вида обычно вызывают вывод предупреждений вроде "Useless use of addition (+) in a void context" (бесполезное сложение в пустом контексте) и "print (...) interpreted as function" (print (...) интерпретируется как функция), если вывод предупреждений включен. Второе сообщение напоминает, что круглые скобки считаются ограничителями списка аргументов, и что все остальное, находящееся за их пределами, не является частью этого списка аргументов. Вот как следует записывать подобные конструкции:

```
print(($foo & 255) + 1, "\n"); # выведет ($foo & 255)+1
```

Анализатор также считает термами конструкции do {} и eval {}, вызовы подпрограмм и методов, конструкторы анонимных массивов и хешей [] и {}, а также анонимный конструктор подпрограмм sub {}.

Оператор «стрелка»

Точно так же, как в C и C++, бинарный оператор -> является инфиксным оператором разыменования. Если правая часть представляет собой индекс массива [...], индекс хеша {...} или список аргументов подпрограммы (...), то левая часть должна быть ссылкой[2] на массив, хеш и подпрограмму, соответственно:

```
$aref->[42]               # разыменование массива
$href->{"corned beef"}    # разыменование хеша
$sref->(1,2,3)            # разыменование подпрограммы
```

В контексте l-значения (допускающем присваивание), когда левая часть не является ссылкой, она должна быть неким адресом, по которому может храниться жесткая ссылка, и в этом случае происходит *самооживление* (*autovivification*) такой ссылки.

```
$aref->[42] = 'Huh!';          # самооживление массива в $aref
$href->{"corned beef"} = 0;    # самооживление хеша в $href
```

В любом случае такое использование создает новый элемент массива или хеша с присвоенным значением. Более подробно об этом (с некоторыми предупреждениями относительно случайного самоооживления) рассказано в главе 8.

Справа от стрелки должны находиться либо скобки того или иного вида, либо вызов некоторого метода. Правая часть должна быть именем метода (или простой скалярной переменной, содержащей имя метода), а левая часть должна вычисляться как объект (связанная с объектом ссылка) или имя класса (т.е. имя пакета):

```
my $yogi = Bear->new("Yogi");      # вызов метода класса
$yogi->swipe('корзина с едой');    # вызов метода объекта
```

Имя метода может быть квалифицировано именем пакета для указания класса, с которого нужно начать поиск метода, либо специальным именем пакета SUPER::, указывающим, что поиск должен быть начат с родительского класса. См. главу 12.

---

[1] Именно поэтому данная проблема будет исправлена в Perl 6. Увы, ее не так-то просто исправить в Perl 5, сохранив работоспособность существующих программ.

[2] Это может быть символическая ссылка, но только когда прагма strict не действует. В противном случае это должна быть жесткая ссылка.

Автоинкрементирование и автодекрементирование

Операторы ++ и -- действуют так же, как в С. Это значит, что, будучи помещенными перед переменной, они инкрементируют или декрементируют переменную, перед тем как вернуть ее значение, а будучи помещенными после нее, они инкрементируют или декрементируют переменную после того, как прочитано ее значение. Например, $a++ инкрементирует значение скалярной переменной $a, возвращая ее значение *перед* выполнением инкрементирования. Аналогично --$b{(/(\w+)/)[0]} декрементирует элемент хеша %b с индексом, равным первому «слову» в установленной по умолчанию для поиска переменной ($_) и возвращает значение, полученное *после* декрементирования.[1] Имейте в виду, что как и язык С, Perl не определяет конкретный момент, когда произойдет инкрементирование или декрементирование. Известно лишь, что оно произойдет до или после возврата значения. Это означает, что попытка дважды применить эти операторы в одном выражении к одной и той же переменной может привести к неожиданным результатам. Избегайте таких выражений:

```
$i = $i++;
print ++$i + $i++;
```

Perl не гарантирует однозначное выполнение такого кода.

В оператор автоинкрементирования встроено еще одно «волшебное» свойство. При инкрементировании числовой переменной или переменной, когда-либо упоминавшейся в числовом контексте, мы получаем обычное приращение на единицу. Если, однако, переменная после присваивания ей значения использовалась только в строковом контексте и имеет значение, отличное от пустой строки и соответствующее шаблону /^[a-zA-Z]*[0-9]*\z/, то выполняется инкрементирование строки с сохранением символов в этом диапазоне и переносом:

```
my $foo;
$foo = "99"; print ++$foo;      # выведет "100"
$foo = "a9"; print ++$foo;      # выведет "b0"
$foo = "Az"; print ++$foo;      # выведет "Ba"
$foo = "zz"; print ++$foo;      # выведет "aaa"
```

Неопределенное значение всегда интерпретируется как числовое и перед инкрементированием приобретает значение 0, поэтому постфиксный оператор автоинкремента вернет 0, а не undef.

---

[1] На самом деле, это не совсем честно. Мы просто хотели проверить вашу внимательность. Вот как работает это выражение. Сначала поиск по шаблону находит первое слово в $_, используя регулярное выражение \w+. Благодаря окружающим его скобкам найденное значение возвращается как список из одного элемента, поскольку поиск по шаблону выполняется в списочном контексте. Списочный контекст предоставляется оператором среза списка, (...)[0], который возвращает первый (и единственный) элемент списка. Это значение используется как ключ для хеша, элемент хеша (значение) декрементируется и возвращается. В общем, сталкиваясь со сложным выражением, анализируйте его в направлении «изнутри – наружу», чтобы установить порядок, в котором происходят события.

На момент написания книги волшебное автоинкрементирование не распространялось на буквы и цифры Юникода, но в будущем это может произойти.

Оператор автодекрементирования волшебным не является.

Возведение в степень

Бинарный оператор ** осуществляет возведение в степень. Обратите внимание, что он связывает аргументы даже сильнее, чем унарный минус, поэтому -2**4 равно -(2**4), а не (-2)**4. Этот оператор реализован с помощью С-функции *pow*(3), работа которой строится на числах с плавающей запятой. Расчет выполняется с использованием логарифмов, поэтому можно возводить и в дробные степени, но иногда получаемые результаты не столь точны, какими могли бы быть при обычном умножении.

Идеографические унарные операторы

Большинству унарных операторов назначены имена (см. раздел «Именованные унарные операторы и операторы проверки файлов» далее в этой главе), но некоторые операторы считаются настолько важными, что заслужили собственное особое символическое представление. Все эти операторы, похоже, связаны с отрицанием. Вините математиков.

Унарный ! осуществляет логическое отрицание, т.е. «не». Вариант логического отрицания с более низким приоритетом – not. Значением отрицаемого операнда становится «истина», или true (1), если операнд есть «ложь», или false, (число 0, строка "0", пустая строка или неопределенное значение), и «ложь», или false, (""), если операнд есть «истина».

Унарный минус (-) осуществляет арифметическое отрицание, если операнд является числом. Если операнд представляет собой идентификатор, то возвращается строка, состоящая из знака «минус», конкатенированного с идентификатором. В иных случаях, если строка начинается со знака «плюс» или «минус», возвращается строка, начинающаяся с противоположного знака. Отсюда, в частности, следует, что -bareword эквивалентно "-bareword".[1] Но если строка начинается любым символом, кроме буквы, «+» или «-», Perl попытается преобразовать строку в число и выполнить арифметическое отрицание. Если строка не может быть преобразована в число, Perl выдаст предупреждение "Argument "the string" isn't numeric in negation (-)" («Аргумент "the string" не является числом в операции арифметического отрицания (–)»).

Унарная тильда (~) осуществляет поразрядное отрицание, т.е. дополнение до единицы. Например, выражение 0666 & ~027 даст в результате 0640. Из определения следует, что этот оператор не вполне переносим, поскольку связан с размером машинного слова. Например, на 32-разрядной машине ~123 равно 4294967172, а на 64-разрядной равно 18446744073709551492. Но об этом читатель уже знает.

[1] Это очень удобно для программирующих на Tk. Для них, собственно, это соглашение и было введено первоначально.

О чем он, возможно, не знает, так это о том, что если аргументом оператора ~ оказывается не число, а строка, то возвращается строка той же длины, но все разряды ее являются дополнениями до единицы. Это быстрый способ «перекинуть» за одно обращение большое количество битов, и способ, к тому же, переносимый, поскольку результат не зависит от размера машинного слова. Далее мы еще расскажем о поразрядных логических операторах, имеющих варианты, ориентированные на строки.

Если коды всех символов в дополняемой строке не превышают значения 256, это же ограничение сохранится и для дополнения. В противном случае все символы будут приведены к 32- или 64-битным значениям, в зависимости от аппаратной архитектуры, и затем дополнены. Так, например, выражение ~"\x{3B1}" даст результат "\x{FFFF_FC4E}" на 32-битной машине и "\x{FFFF_FFFF_FFFF_FC4E}" на 64-битной.

Унарный плюс (+) не оказывает никакого семантического действия – даже на строки. Его синтаксическая польза состоит в отделении имени функции от выражения в скобках, которое иначе интерпретировалось бы как полный список аргументов функции. (См. примеры раздела «Термы и списочные операторы» данной главы.) Можно представить себе это и так, что + отрицает действие скобок по превращению префиксных операторов в функции.

Унарный оператор \ создает ссылку на то, что следует за ним. Применение его к списку создает список ссылок. Подробности можно найти в разделе «Оператор обратной косой черты» главы 8. Не путайте действие этого оператора с действием обратной косой черты внутри строки, хотя в обоих случаях присутствует некоторая идея отрицания, защищающего последующий объект от интерпретации. Это сходство не является совершенно случайным.

Операторы связывания

Бинарный оператор =~ связывает строковое выражение с поиском по шаблону, подстановкой или транслитерацией (вольно называемой трансляцией). Если не использовать этот оператор, то поиск или замена осуществляются в строке, содержащейся в переменной $_ (переменной по умолчанию). Строка, подлежащая связыванию, помещается в левую часть, а сам оператор – справа от нее. Возвращаемое значение указывает на успех или неудачу оператора справа, поскольку оператор связывания, в сущности, никаких самостоятельных действий не выполняет. Исключение составляет использование модификатора /r в операции подстановки (s///) или транслитерации (y///, tr///) – в этом случае возвращается копия измененной строки. Поведение в списочном контексте зависит от конкретного оператора.

Если правый аргумент представляет собой выражение, а не оператор поиска по шаблону, замены или транслитерации, то на этапе выполнения это выражение будет интерпретироваться как шаблон, по которому должен быть проведен поиск. Например, $_ =~ $pat эквивалентно $_ =~ /$pat/. Это менее эффективно, чем явный поиск, поскольку шаблон должен проверяться и, возможно, перекомпилироваться при каждом вычислении выражения. Избежать повторной компиляции можно путем предварительной компиляции исходного шаблона с использованием оператора цитирования регулярного выражения, qr//.

Бинарный оператор !~ аналогичен =~, с тем отличием, что возвращается логическое отрицание результата. Использование бинарного оператора !~ с модификатором /r

для неразрушающей подстановки или транслитерации вызывает синтаксическую ошибку. Во всем остальном следующие выражения действуют одинаково:

```
$string !~ /pattern/
!( $string =~ /pattern/ )
not $string =~ /pattern/
```

Мы сказали, что возвращаемое значение сигнализирует об успешности операции, но есть разные типы успеха. Если не используется модификатор /r для принудительного возврата результатов поиска, подстановка возвращает *число* найденных совпадений, как и транслитерация. (На практике оператор транслитерации часто применяют для подсчета символов.) Поскольку любой ненулевой результат является истиной, то все работает. Наиболее впечатляющий пример «истинного» значения дает присваивание шаблона списку: в списочном контексте поиск по шаблону может возвращать подстроки, соответствующие круглым скобкам, имеющимся в шаблоне. И снова, согласно правилам присваивания списку, сама операция присваивания возвращает истину, если что-либо было найдено и присвоено, и ложь в противном случае. Поэтому иногда можно встретить код такого вида:

```
if (my ($k,$v) = $string =~ m/(\w+)=(\w*)/ ) {
    print "КЛЮЧ $k ЗНАЧЕНИЕ $v\n";
}
```

Разберем этот пример. Оператор =~ имеет больший приоритет, чем =, поэтому сначала выполняется =~. Оператор =~ связывает $string с шаблоном в правой части, который ищет в строке подстроку, похожую на *КЛЮЧ=ЗНАЧЕНИЕ*. Этот оператор находится в списочном контексте, поскольку расположен в правой части присваивания списку. Если поиск успешен, возвращается список, который нужно присвоить $k и $v, новым переменным, созданным посредством объявления my. Само присваивание списку находится в скалярном контексте, поэтому оно возвращает 2 — число значений в правой части присваивания. А 2 оказывается истиной, поскольку наш скалярный контекст является также логическим. Если соответствия не найдены, значения не присваиваются, и возвращается 0, т.е. ложное значение.

Подробности о замысловатостях шаблонов читайте в главе 5.

Мультипликативные операторы

В Perl имеются операторы * (умножение), / (деление) и % (взятие по модулю), аналогичные операторам языка C. Операторы * и / действуют точно так, как можно предположить: перемножают и делят свои операнды. Деление осуществляется с вещественной точностью, если не используется прагма integer, bigint, bigrat или bignum. Оператор % преобразует свои операнды в целые числа, а затем находит остаток от деления целых чисел. (Однако при необходимости он осуществляет это деление с вещественной точностью, поэтому на большинстве 32-разрядных машин операнды могут иметь размер до 15 цифр.) Возьмем для примера операнды с именами $a и $b. В отличие от математической функции вычисления остатка, оператор % в Perl определяет пилообразную функцию на множестве действительных чисел. На каждом интервале между соседними кратными делителями она сначала резко возрастает почти до первого из них, а затем плавно падает до 0. Эта функция не симметрична относительно начала координат, но сохраняет пилообразную форму. Математически $a % $b можно выразить как:

```
use POSIX;
$a % $b == $a - ( POSIX::floor($a / $b) * $b )
```

Если в области видимости имеется директива use integer, то % предоставляет прямой доступ к оператору взятия по модулю в том виде, в каком он реализован в вашем компиляторе C. Этот оператор не вполне корректно определен для отрицательных операндов, но выполняется быстрее.

Бинарный оператор x является оператором повторения. Фактически в нем два разных оператора. В скалярном контексте он возвращает конкатенированную строку, состоящую из левого операнда, повторенного число раз, заданное правым операндом. (Для обратной совместимости он делает это и в списочном контексте, если левый аргумент не заключен в круглые скобки.)

```
print '-' x 80;                          # выведет строку дефисов
print "\t" x ($tab/8), " " x ($tab%8);  # замена пробелов табуляцией
```

В списочном контексте, если левый операнд заключен в круглые скобки или список сформирован, как qw/*STRING*/, то x действует как репликатор списка, а не репликатор строки. Это удобно для инициализации массива неопределенной длины одинаковыми значениями:

```
my @ones = (1) x 80;    # список из 80 единиц
@ones = (5) x @ones;    # установить все элементы равными 5
```

Аналогично оператор x можно использовать для инициализации срезов массивов и хешей:

```
my %hash;
my @keys = qw(perls before swine);
@hash{@keys} = ("") x @keys;
```

Если вам это непонятно, обратите внимание, что @keys выступает как список в левой части присваивания и как скалярная величина (возвращающая размер массива) в правой части присваивания. Предыдущий пример оказывает на %hash такое же воздействие, как:

```
$hash{perls}  = "";
$hash{before} = "";
$hash{swine}  = "";
```

Аддитивные операторы

Как ни странно, в Perl есть и обычные операторы + (сложение) и - (вычитание). Оба оператора при необходимости преобразуют свои аргументы из строк в числа и возвращают числовой результат.

Кроме того, в Perl имеется оператор . для конкатенации строк. Например:

```
my $almost = "Fred" . "Flintstone";  # возвращает FredFlintstone
```

Заметьте, что Perl не помещает между конкатенируемыми строками пробел. Если пробел необходим или нужно конкатенировать более двух строк, можно использовать оператор join, описываемый в главе 27. Чаще всего, однако, конкатенацию выполняют неявным образом, заключая строку в двойные кавычки:

```
my $fullname = "$firstname $lastname";
```

Операторы сдвига

Операторы поразрядного сдвига (<< и >>) возвращают значение левого аргумента, сдвинутое влево (<<) или вправо (>>) на число разрядов, заданное правым аргументом. Аргументы должны быть целыми числами. Например:

```
1 << 4;        # вернет 16
32 >> 4;       # вернет 2
```

Будьте бдительны. Результат применения этих операторов к большим (или отрицательным) числам может зависеть от разрядности вашей машины. Избежать этого ограничения можно с помощью прагмы `bigint`.

```
use v5.14;
say 500 <<  20;    # выведет 524288000
say 500 << 200;    # выведет (только) 128000

use bigint;
say 500 << 200;
803469022129495137770981046170581301261101496891396417650688000
```

Именованные унарные операторы и операторы проверки файлов

Некоторые из «функций», описываемых в главе 27, являются в действительности унарными операторами. В табл. 3.2 перечислены все именованные унарные операторы.

Таблица 3.2. Именованные унарные операторы

-X (проверка файлов)	eval	gmtime	prototype	sleep
abs	exists	hex	quotemeta	sqrt
alarm	exit	int	rand	srand
caller	exp	keys	readdir	stat
chdir	fc	lc	readline	state
chomp	fileno	lcfirst	readlink	study
chop	getc	length	readpipe	tell
chr	getgrgid	local	ref	telldir
chroot	getgrnam	localtime	reset	tied
close	gethostbyname	lock	rewinddir	uc
closedir	getnetbyname	log	rmdir	ucfirst
cos	getpeername	lstat	scalar	umask
dbmclose	getpgrp	my	sethostent	undef
defined	getprotobyname	oct	setnetent	untie
delete	getpwnam	ord	setprotoent	values
do	getpwuid	our	setservent	write
each	getsockname	pop	shift	любая ($) подпрограмма
eof	glob	pos	sin	

В отличие от списочных операторов, унарные операторы имеют более высокий приоритет, чем некоторые бинарные. Например:

```
sleep 4 | 3;
```

приостановит выполнение не на 7 секунд, а на 4, а затем возьмет результат выполнения sleep (обычно 0), и выполнит операцию поразрядного ИЛИ с числом 3, как если бы скобки были расставлены так:

```
(sleep 4) | 3;
```

Сравните с:

```
print 4 | 3;
```

Этот оператор *выполняет* логическое ИЛИ 4 и 3 (7 в данном случае), перед выводом, как если бы стояли скобки:

```
print (4 | 3);
```

Дело в том, что print представляет собой списочный оператор, а не простой унарный. Если запомнить, какие операторы являются списочными, будет несложно отличить унарный оператор от списочного. Если возникают сомнения, всегда можно воспользоваться круглыми скобками, чтобы превратить унарный оператор в функцию. Помните: что выглядит как функция, функцией и является.

Еще одно любопытное обстоятельство, связанное с именованными унарными операторами, заключается в том, что многие из них по умолчанию работают с переменной $_, если не передать им никакого аргумента. Однако если аргумент опущен, а лексема, следующая за унарным оператором, похожа на начало аргумента, Perl придет в замешательство, поскольку ожидает появления терма. Когда анализатор лексем Perl встречает один из символов, перечисленных в табл. 3.3, то возвращает различные типы лексем в зависимости от того, что он рассчитывает получить – терм или оператор.

Таблица 3.3. Двусмысленные символы

Символ	Оператор	Терм
+	Сложение	Унарный плюс
-	Вычитание	Унарный минус
*	Умножение	*typeglob
/	Деление	/шаблон/
<	Меньше, сдвиг влево	<HANDLE>, <<END
.	Конкатенация	.3333
?	?:	?шаблон?
%	Взятие по модулю	%хеш
&	&, &&	&подпрограмма

Типичная глупая ошибка выглядит так:

```
next if length < 80;
```

В этом случае < кажется анализатору началом символа ввода <> (термом), а не оператором «меньше», который вам нужен. Исправить эту ситуацию в Perl и одно-

временно сохранить его патологическую эклектичность решительно невозможно. Если вы невероятно ленивы и не можете заставить себя набрать два символа конструкции $_, воспользуйтесь одним из следующих способов:

```
next if length() < 80;
next if (length) < 80;
next if 80 > length;
next unless length >= 80;
```

Если ожидается терм, то знак «минус», за которым следует одна буква, всегда рассматривается как *оператор проверки файла*. Оператор проверки файла представляет собой унарный оператор, который принимает один аргумент, являющийся именем файла или его дескриптором, и проверяет, обладает ли этот файл некоторым свойством. Если аргумент опущен, оператор проверяет $_. Исключение: оператор -t, который проверяет STDIN. Если в документации не указано иное, оператор проверки файла возвращает 1 при выполнении условия и "" при его невыполнении, либо неопределенное значение, если файл не существует или недоступен по каким-то другим причинам. Реализованные в настоящее время операторы проверки файлов перечислены в табл. 3.4.

Таблица 3.4. Операторы проверки файлов

Оператор	Значение
-r	Файл доступен для чтения текущему пользователю или группе (effective UID/GID)
-w	Файл доступен для записи текущему пользователю или группе (effective UID/GID)
-x	Файл доступен для выполнения текущему пользователю или группе (effective UID/GID)
-o	Файл принадлежит текущему пользователю (effective UID)
-R	Файл доступен для чтения реальному пользователю или группе (real UID/GID)
-W	Файл доступен для записи реальному пользователю или группе (real UID/GID)
-X	Файл доступен для выполнения реальному пользователю или группе (real UID/GID)
-O	Файл принадлежит реальному пользователю (real UID)
-e	Файл существует
-z	Файл имеет нулевую длину
-s	Файл имеет ненулевую длину (возвращает размер)
-f	Обычный файл
-d	Каталог
-l	Файл является символической ссылкой
-p	Файл является именованным каналом (FIFO)
-S	Файл является сокетом
-b	Специальный файл блочного устройства
-c	Специальный файл символьного устройства
-t	Дескриптор файла связан с терминалом
-u	Для файла установлен бит setuid

Таблица 3.4 (продолжение)

Оператор	Значение
-g	Для файла установлен бит setgid
-k	Для файла установлен «липкий бит» (sticky bit)
-T	Файл является текстовым
-B	Файл является двоичным (проверка, обратная -T)
-M	Время, прошедшее с момента последней модификации файла до запуска сценария, в днях
-A	Время, прошедшее с момента последнего обращения к файлу до запуска сценария, в днях
-C	Время, прошедшее с момента последнего изменения индексного дескриптора до запуска сценария, в днях

На эти операторы не распространяется правило «если выглядит как функция...», описанное выше. То есть наличие открывающей скобки после оператора не влияет на то, как будут интерпретироваться его аргументы. Это означает, например, что выражение -f($file)."".bak" эквивалентно выражению -f "$file.bak". Чтобы отделить оператор проверки файла от следующего за ним кода, достаточно просто заключить его в круглые скобки (разумеется, это необходимо, только если в выражении присутствуют операторы, имеющие более высокий приоритет, чем унарные операторы):

```
-s($file) + 1024   # возможно, ошибка; то же, что и -s($file + 1024)
(-s $file) + 1024  # правильно
```

Обратите внимание, что -s/a/b/ не производит подстановку с отрицанием. Однако -exp($foo) работает, как должно: только одиночные буквы после минуса трактуются как проверка файла.

Интерпретация операторов прав доступа к файлам -r, -R, -w, -W, -x и -X основывается исключительно на свойствах файла и идентификаторах пользователя и группы. Могут быть и другие причины, по которым не удается читать, записывать или выполнять файл. Например, в системе применяются списки управления доступом (Access Control Lists, ACL), и вы не включены в список.[1] Заметьте также, что для суперпользователя -r, -R, -w и -W всегда возвращают 1, а -x и -X возвращают 1, если установлен хоть один бит исполнения. Поэтому в сценариях, работающих с полномочиями суперпользователя, может потребоваться выполнить stat, чтобы определить действительный режим файла, либо временно установить другой идентификатор пользователя. Остальным операторам проверки файлов безразлично, кто вы такой. Проверку того, является ли файл обычным, может выполнять любой:

```
while (<>) {
    chomp;
    next unless -f $_;      # пропустить "особые" файлы
    ...
}
```

[1] Однако можно изменить стандартную семантику с помощью прагмы filetest. См. главу 29.

Ключи -T и -B действуют следующим образом. Участок файла примерно в объеме первого блока исследуется на наличие необычных символов, таких как коды управления или байты с установленным старшим битом (не похожие на UTF-8). Если необычных байтов более трети, такой файл считается двоичным, в противном случае – текстовым. Кроме того, двоичным считается любой файл, в первом блоке которого встретился символ ASCII NUL (\0). Если -T или -B применяется к дескриптору файла, то исследуется не первый блок файла, а текущий буфер ввода (стандартный I/O или «stdio»). Оба оператора, -T и -B, возвращают истину для пустого файла или файла, указатель которого находится в EOF – конце файла, если проверяется дескриптор. Поскольку Perl требуется прочесть файл, чтобы выполнить проверку -T, применение последней со специальными файлами, которые могут вызвать зависание или доставить другие неприятности, нежелательно. Поэтому чаще всего нужно сначала выполнить проверку с ключом -f, например:

```
next unless -f $file && -T $file;
```

Если какой-либо из проверок файлов (или операторам stat или lstat) передается специальный дескриптор файла, состоящий из одного символа подчеркивания, то используется структура *stat* от предыдущей проверки файла (или оператора stat), что позволяет избежать нового системного вызова. (Это не действует при наличии директивы use filetest и для оператора -t, и нужно помнить, что lstat и -l оставляют в структуре stat значения, относящиеся к символической ссылке, а не к действительному файлу. Аналогично, -l _ всегда возвращает «ложь» после обычного stat.)

Вот несколько примеров:

```
print "Доступен для обработки.\n" if -r $a || -w _ || -x _

stat($filename);
print "Доступен для чтения\n"   if -r _
print "Доступен для записи\n"   if -w _
print "Исполняемый\n"           if -x _
print "Установлен бит setuid\n" if -u _
print "Установлен бит setgid\n" if -g _
print "Установлен бит sticky\n" if -k _
print "Текстовый\n"             if -T _
print "Двоичный\n"              if -B _;
```

Время существования файла возвращается операторами -M, -A и -C в днях (включая дробную часть суток) с момента последней модификации, обращения или изменения индексного дескриптора до запуска сценария, хранящегося в переменной $^T ($BASETIME). Если файл был изменен после начала выполнения сценария, возвращается отрицательное время. Отметьте, что значения времени в большинстве случаев (в среднем 86399 из 86400) являются дробными, поэтому проверка равенства целому числу без использования функции int обычно оказывается бесполезной. Примеры:

```
next unless -M $file > .5;      # файлы старше 12 часов
&newfile if -M $file < 0,       # файл новее, чем процесс
&mailwarning if int(-A) == 90;  # обращение к файлу ($_) ровно 90 дней назад
```

Чтобы установить время запуска сценария равным текущему времени, достаточно выполнить:

```
$^T = time;
```

Начиная с Perl v5.10 синтаксис дополнился удобной возможностью комбинировать операторы проверки файлов. В результате конструкция `-f -w -x $file` равноценна `-x $file && -w _ && -f _`.

Операторы сравнения

В Perl имеется два класса операторов сравнения. Один класс выполняет операции сравнения со строковыми величинами, другой с числовыми, как показано в табл. 3.5.

Таблица 3.5. Операторы сравнения

Числа	Строки	Значение
>	gt	Больше
>=	ge	Больше или равно
<	lt	Меньше
<=	le	Меньше или равно

Эти операторы возвращают 1 в качестве истины и "" – как ложь. Заметьте, что операторы сравнения неассоциативны, что означает синтаксическую ошибку в выражении `$a < $b < $c`.

В отсутствие объявлений региональных настроек сравнение строк основывается на значениях кодов символов Юникода. Если же такое объявление присутствует, используется соответствующая указанным региональным настройкам схема упорядочения. Устаревшие механизмы сравнения, основанные на региональных настройках, не всегда хорошо взаимодействуют с механизмами сравнения Юникода, предоставляемыми модулями `Unicode::Collate` и `Unicode::Collate::Locale`. Поэтому лучше использовать модули, а не региональные настройки. Порядок кодов символов не совпадает с алфавитным порядком следования символов, за исключением символов ASCII, поэтому строковые операторы в Perl производят результаты в алфавитном порядке для символов формата ASCII, но не для произвольных текстов вообще.

Операторы равенства

Операторы равенства, перечисленные в табл. 3.6, во многом похожи на операторы сравнения.

Таблица 3.6. Операторы равенства

Числа	Строки	Значение
==	eq	Равно
!=	ne	Не равно
<=>	cmp	Сравнение, результат со знаком
--	--	Интеллектуальное сопоставление

Операторы проверки равенства и неравенства возвращают 1 в качестве значения «истина» и "" в качестве значения «ложь» (как и операторы сравнения). Операторы <=> и cmp возвращают -1, если левый операнд меньше правого, 0 при их равенстве, и +1, если левый операнд больше правого. Хотя операторы равенства похожи на операторы сравнения, они имеют более низкий приоритет, поэтому запись `$a < $b <=> $c < $d` синтаксически допустима.

По причинам, очевидным для каждого видевшего фильм «Звездные войны», оператор <=> называют также оператором «космического корабля» (spaceship).

Оператор ~~ описывается в следующем разделе.

Оператор интеллектуального сопоставления

Впервые представленный в Perl v5.10.1,[1] бинарный оператор ~~ (двойная тильда) выполняет «интеллектуальное сопоставление» своих операндов. В большинстве случаев он неявно используется в конструкциях when, хотя не все ветви when используют оператор интеллектуального сопоставления. От других операторов Perl этот оператор отличает его способность к рекурсивному выполнению.

Другая его особенность в том, что остальные операторы предполагают определенный контексту (обычно строковой или числовой) для своих операндов и автоматически выполняют преобразование операндов в нужный контекст. Оператор интеллектуального сопоставления *выводит* контекст из фактических типов операндов и на основе типов выбирает наиболее подходящий механизм сравнения.

Оператор ~~ сравнивает свои операнды «полиморфически», выбирая способ сравнения на основе из их фактических типов (число, строка, массив, хеш, и так далее). Подобно операторам определения равенства, с которыми он совпадает по уровню приоритета, оператор ~~ возвращает 1 в качестве значения «истина» и "" в качестве значения «ложь». Как и в операторе связывания =~, правый аргумент этого оператора считается шаблоном, которому может соответствовать, либо не соответствовать левый аргумент. Однако, понятие «шаблон» здесь используется в широком смысле: практически любое значение может играть роль шаблона или списка шаблонов.

Оператор ~~ часто лучше читать, как «соответствует» или «соответствует любому из», потому что левый операнд проверяется на соответствие правому операнду (или некоторой части правого операнда).

Поведение оператора интеллектуального сопоставления зависит от типов аргументов, как определено в табл. 3.7. В первой колонке таблицы перечислены типы, определяющие поведение оператора интеллектуального сопоставления. Но, поскольку сначала определяется тип правого операнда, и только потом левого, таблица отсортирована по типу правого операнда.

Оператор интеллектуального сопоставления автоматически разыменовывает все несвязанные ссылки на хеши или массивы, поэтому таким ссылкам соответствуют строки *HASH* и *ARRAY*. Связанным ссылкам соответствуют строки *Object*. При сопоставлении хешей всегда исследуются только ключи, и никогда – значения.

---

[1] В версии v5.10.0 этот оператор действует иначе в некоторых пограничных случаях, но в этом нет ничего страшного, потому что сейчас вы используете версию не ниже v5.14, правда?

Пример в колонке «Аналог» не всегда точно соответствует фактической реализации. Например, оператор интеллектуального сопоставления использует укороченную схему вычислений везде, где только возможно, а команда grep – нет. Кроме того, команда grep в скалярном контексте возвращает количество совпадений, а оператор ~~ возвращает только true или false.

В отличие от большинства операторов, оператор интеллектуального сопоставления интерпретирует значение undef по-особенному:

```
my @array = (1, 2, 3, undef, 4, 5);
say "некоторые элементы не определены" if undef ~~ @array;
```

Каждый операнд исследуется в модифицированном скалярном контексте, в результате такой модификации массивы и хеши передаются оператору по ссылкам, которые он, в свою очередь, разыменовывает. Оба элемента в каждой паре суть одно и то же:

```
my %hash = (red    => 1, blue   => 2, green  => 3,
            orange => 4, yellow => 5, purple => 6,
            black  => 7, grey   => 8, white  => 9);

my @array = qw(red blue green);

say "некоторые элементы массива совпадают с ключами хеша" if @array ~~ %hash;
say "некоторые элементы массива совпадают с ключами хеша" if \@array ~~ \%hash;

say "цвет red присутствует в массиве" if "red" ~~ @array;
say "цвет red присутствует в массиве" if "red" ~~ \@array;

say "некоторые ключи оканчиваются на e" if /e$/ ~~ %hash;
say "некоторые ключи оканчиваются на e" if /e$/ ~~ \%hash
```

Таблица 3.7. Поведение оператора интеллектуального сопоставления

Слева	Справа	Описание	Аналог (но вычисляется в логическом контексте)
Any	*undef*	Проверяет, является ли произвольное значение *Any* неопределенным значением	`!defined Any`
Any	*Object*	Вызывает перегруженную реализацию оператора ~~ в *Object* или завершается ошибкой	
HASH	*CODE*	Подпрограмма возвращает true для всех ключей хеша *HASH*[a]	`!grep { !CODE->($_) } keys HASH`
ARRAY	*CODE*	Подпрограмма возвращает true для всех элементов массива *ARRAY*	`!grep { !CODE->($_) } ARRAY`
Any	*CODE*	Подпрограмма возвращает true для *Any*	`CODE->(Any)`
HASH1	*HASH2*	Оба хеша обладают одним и тем же набором ключей	`keys HASH1 == grep { exists HASH2->{$_} } keys HASH1`
ARRAY	*HASH*	Хотя бы один из элементов массива *ARRAY* совпадает с одним из ключей хеша *HASH*	`grep { exists HASH->{$_} } ARRAY`

Слева	Справа	Описание	Аналог (но вычисляется в логическом контексте)
Regexp	*HASH*	Любой из ключей *HASH* соответствует шаблону *Regexp*	`grep { /Regexp/ } keys HASH`
undef	*HASH*	Всегда false (undef не может быть ключом)	`0 == 1`
Any	*HASH*	Наличие ключа *Any* в хеше *HASH*	`exists HASH->{Any}`
HASH	*ARRAY*	Хотя бы один из элементов массива *ARRAY* совпадает с одним из ключей хеша *HASH*	`grep { exists HASH->{$_} } ARRAY`
ARRAY1	*ARRAY2*	Рекурсивно сопоставляет парные элементы массивов *ARRAY1* и *ARRAY2*[b]	`(ARRAY1[0] ~~ ARRAY2[0]) && (ARRAY1[1] ~~ ARRAY2[1]) &&`
Regexp	*ARRAY*	Любой элемент массива ARRAY соответствует шаблону *Regexp*	`grep { /Regexp/ } ARRAY`
undef	*ARRAY*	Проверяет наличие undef в массиве *ARRAY*	`grep { !defined } ARRAY`
Any	*ARRAY*	Выполняется интеллектуальное сопоставление с каждым элементом массива *ARRAY*[c]	`grep { Any ~~ $_ } ARRAY`
HASH	*Regexp*	Любой из ключей *HASH* соответствует шаблону *Regexp*	`grep { /Regexp/ } keys HASH`
ARRAY	*Regexp*	Любой элемент массива ARRAY соответствует шаблону *Regexp*	`grep { /Regexp/ } ARRAY`
ANY	*Regexp*	Обычный поиск по шаблону	`Any =~ /Regexp/`
Object	*Any*	Вызывает перегруженную реализацию оператора ~~ в *Object* или используется сопоставление по умолчанию	
Any	*Num*	Числовое сравнение	`Any == Num`
Num	*numlike*[d]	Числовое сравнение	`Any == numlike`
undef	*Any*	Проверяет, является ли произвольное значение *Any* неопределенным значением	`!defined Any`
Any	*Any*	Строковое сравнение	`Any eq Any`

[a] Пустые хеши и массивы автоматически признаются соответствующими.

[b] То есть каждый элемент из одного массива сопоставляется с соответствующим элементом из другого массива с помощью того же оператора интеллектуального сопоставления (см. следующую сноску).

[c] При обнаружении циклической ссылки сопоставление сводится к сравнению ссылок.

[d] Либо число, либо строка, которая выглядит как число.

Два массива считаются соответствующими, если каждый элемент первого массива рекурсивно соответствует (в терминах интеллектуального сопоставления) элементу второго массива с тем же индексом:

```
my @little = qw(красный синий зеленый);
my @bigger = ("красный", "синий", [ "оранжевый", "зеленый" ] );
if (@little ~~ @bigger) {      # истина!
    say "малое содержится в большом";
}
```

Поскольку оператор интеллектуального сопоставления при необходимости производит рекурсивное сопоставление с вложенными массивами, следующий фрагмент также сообщит, что элемент "красный" присутствует в массиве:

```
my @array = qw(красный синий зеленый);
my $nested_array = [[[[[[[ @array ]]]]]]];
say "красный в массиве" if "красный" ~~ $nested_array;
```

Если массивы соответствуют один другому, они являются глубокими копиями друг друга, как видно из этого примера:

```
my @a = (0, 1, 2, [3, [4, 5], 6], 7);
my @b = (0, 1, 2, [3, [4, 5], 6], 7);

if (@a ~~ @b && @b ~~ @a) {
    say "a и b - глубокие копии друг друга";
}
elsif (@a ~~ @b) {
    say "a присутствует в b",
}
elsif (@b ~~ @a) {
    say "b присутствует в a",
}
else {
    say "a и b не соответствуют друг другу"
}
```

Если выполнить эту программу, она выведет:

```
a и b - глубокие копии друг друга
```

Если выполнить инструкцию `$b[3] = 4`, тогда пример выведет "b присутствует в a", потому что в соответствующей позиции в массиве `@a` содержится вложенный массив, содержащий (на каком-то уровне) 4.

Интеллектуальное сопоставление одного хеша с другим позволяет выяснить, содержат ли они одинаковые ключи, ни больше, ни меньше. Этот факт можно использовать, например, чтобы определить наличие полей с одинаковыми именами в двух записях, без учета значений этих полей. Например:

```
use v5.10;
sub make_dogtag {
    state $REQUIRED_FIELDS = { name=>1, rank=>1, serial_num=>1 };

    my ($class, $init_fields) = @_;

    die "Запись должна содержать (только) название, ранг и серийный номер"
        unless $init_fields ~~ $REQUIRED_FIELDS;


}
```

Или, если другие поля допустимы, но не требуются, используйте сопоставление *ARRAY ~~ HASH*:

```
use v5.10;
sub make_dogtag {
    state $REQUIRED_FIELDS = { name=>1, rank=>1, serial_num=>1 },
```

```
    my ($class, $init_fields) = @_;

    die " Запись должна содержать (минимум) название, ранг и серийный номер
        unless [keys %{$init_fields}] ~~ $REQUIRED_FIELDS;


  }
```

Интеллектуальное сопоставление чаще всего используется неявно, в операторе given/when. См. раздел «Оператор given» в главе 4.

Интеллектуальное сопоставление объектов

Чтобы избежать использования базового представления объекта, когда правым операндом оператора интеллектуального сопоставления является объект, не перегружающий оператор ~~, Perl возбуждает исключение "Smart matching a non-overloaded object breaks encapsulation" (Интеллектуальное сопоставление с объектом, не имеющим перегруженной версии оператора, нарушает принцип инкапсуляции). Поэтому нет никакого способа проверить наличие чего-либо «в» объекте. Следующие выражения недопустимы, если объект не содержит реализацию перегруженного оператора ~~:

```
 %hash ~~ $object
    42 ~~ $object
"fred" ~~ $object
```

Однако есть возможность изменить способ сопоставления с объектом, реализовав перегрузку оператора ~~. Это позволит расширить обычную семантику оператора. Особенности перегрузки оператора ~~ в объектах описываются в главе 13.

Объект может выступать левым операндом в интеллектуальном сопоставлении, но в этом мало смысла. Правила интеллектуального сопоставления имеют преимущества перед перегруженным оператором, поэтому, даже если объект в левом операнде имеет перегруженную версию оператора ~~, Perl ее игнорирует. Если левый операнд не содержит реализацию перегруженного оператора, происходит возврат к числовому или строковому сравнению, в зависимости от того, что вернет оператор ref. То есть:

```
$object ~~ X
```

не вызовет перегруженный метод с аргументом *X*. Вместо этого решение будет приниматься в соответствии с табл. 3.7. В зависимости от типа операнда *X*, перегруженный метод может вызываться или не вызываться. Для простых строк или чисел сопоставление сведется к следующему:

```
$object ~~ $number ref($object) == $number
$object ~~ $string ref($object) eq $string
```

Например, следующий фрагмент сообщит, что «попахивает модулем IO»:

```
use IO::Handle;
my $fh = IO::Handle->new();
if ($fh ~~ /\bIO\b/) {
    say "попахивает модулем IO";
}
```

Это объясняется тем, что `$fh` интерпретируется как строка вида `"IO::Handle=GLOB(0x8039e0)"`, в которой и производится поиск соответствия шаблону.[1]

Операторы поразрядного действия

Как и в C, в Perl есть поразрядные операторы И, ИЛИ, исключающее ИЛИ (XOR) и НЕ: `&`, `|`, `^` и уже знакомый `~`. Взяв на себя труд изучить таблицу в начале этой главы, читатель мог обнаружить, что поразрядное И имеет более высокий приоритет, чем остальные поразрядные операторы, но мы смошенничали и объединили их в нашем изложении.

Эти операторы по-разному действуют на числа и строки. (Один из немногих случаев, когда Perl учитывает разницу между ними.) Если оба операнда – числа (или использовались как числа), они преобразуются в целые числа и поразрядная операция производится над двумя числами. Эти числа имеют гарантированную длину 32 разряда, но на некоторых машинах длина может составлять 64 разряда. Дело в том, что существует произвольное ограничение, налагаемое архитектурой машины. Преодолеть это ограничение можно с помощью прагмы `bigint`.

Если оба операнда суть строки (и не использовались как числа с момента присваивания им значений), операторы выполняют поразрядные операции над соответствующими разрядами двух строк. В этом случае не действует произвольное ограничение, поскольку для размера строк произвольного ограничения нет. Если одна строка длиннее, считается, что более короткая строка имеет в конце достаточное число нулевых разрядов, чтобы компенсировать разницу.

Например, если выполнить И над двумя строками:

```
"123.45" & "234.56"
```

то получится новая строка:

```
"020.44"
```

Но если выполнить И над строкой и числом:

```
"123.45" & 234.56
```

то строка сначала преобразуется в число, что дает:

```
123.45 & 234.56
```

Затем числа преобразуются в целые:

```
123 & 234
```

и в результате получается 106. Обратите внимание, что все битовые строки истинны (если только не равны строке `"0"`). Это значит, что если требуется определить, оказался ли какой-то байт ненулевым, то вместо:

```
if ( "fred" & "\x01\x02\x03\x04" ) {   }
```

---

[1] Однако не следует использовать этот прием на практике. В будущем такое поведение наверняка изменится, чтобы быть ближе к семантике Perl 6, где тип правого аргумента определяет, как будет вести себя объект слева (как строка или число). Поэтому пока просто воздержитесь от выражений с объектами в левой части оператора интеллектуального сопоставления.

нужно записать:

```
if ( ("fred" & "\x01\x02\x03\x04") =~ /[^\0]/ ) {
```

Логические операторы (короткого пути) в стиле C

Как и в C, в Perl имеются операторы && (логическое И) и || (логическое ИЛИ). В Perl также имеется вариант логического оператора || – оператор //, определенное ИЛИ. Они выполняют вычисления слева направо (при этом && имеет несколько больший приоритет, чем || и //), проверяя утверждения на истинность. Эти операторы, перечисленные в табл. 3.8, известны как операторы короткого пути, поскольку определяют истинность утверждения, вычисляя как можно меньшее число операндов. Например, если левый операнд оператора && имеет значение «ложь», то правый операнд не вычисляется вообще, поскольку, вне зависимости от его значения, результат выражения имеет значение «ложь».

Таблица 3.8. Логические операторы

Пример	Название	Результат
`$a && $b`	И	`$a`, если `$a` – `false`, иначе – `$b`
`$a || $b`	ИЛИ	`$a`, если `$a` – `true`, иначе – `$b`
`$a // $b`	определенное ИЛИ	`$a`, если `$a` – определено, иначе – `$b`
`$a and $b`	низкоприоритетное И	`$a`, если `$a` – `false`, иначе – `$b`
`$a or $b`	низкоприоритетное ИЛИ	`$a`, если `$a` – `true`, иначе – `$b`
`$a xor $b`	низкоприоритетное исключающее ИЛИ	`true`, если только один операнд, `$a` или `$b`, имеет значение `true`, иначе – `false`

Короткий путь не просто экономит время, но часто используется для управления порядком вычислений. Например, в программах на Perl часто встречается такая идиома:

```
open(FILE, "<", "некий_файл") || die "Невозможно открыть некий_файл: $!\n";
```

В данном случае Perl сначала вычисляет функцию open. Если ее значение истинно (*некий_файл* был успешно открыт), выполнение функции die не является обязательным и пропускается. Можно прочесть это буквально: «Открыть некий файл или умереть!»

Оператор // удобно использовать с функциями, которые сообщают об ошибке, возвращая значение undef. Например:

```
my $pid = fork() // die "Невозможно запустить дочерний процесс. $!";
if ($pid) {
    # здесь продолжает работу родительский процесс
    ...
    wait $pid;
} else {
    # здесь продолжает работу дочерний процесс
    ...
    exit;
}
```

Его также удобно применять для определения отсутствующих значений в хеше. Следующее выражение вернет "DEFAULT", если искомый ключ отсутствует в хеше или имеет неопределенное значение:

```
$value = $hash{$key} // "DEFAULT";
```

Операторы && и || отличаются от имеющихся в C тем, что возвращают не 0 или 1, а последнее вычисленное значение. В случае || это приводит к тому приятному результату, что можно выбрать первое из ряда скалярных значений, которое окажется истинным. В результате мы получаем весьма переносимый способ поиска домашнего каталога пользователя следующего вида:

```
my $home = $ENV{HOME}
    || $ENV{LOGDIR}
    || (getpwuid($<))[7]
    || die "Да ты бездомный!\n";
```

С другой стороны, поскольку левый аргумент всегда вычисляется в скалярном контексте, нельзя использовать || при выборе для присваивания одного из двух составных объектов:

```
@a = @b || @c,          # Получится не то, что нужно, потому что
@a = scalar(@b) || @c;  # в действительности будет это.
@a = @b ? @b : @c;      # А это работает прекрасно
```

Perl предлагает также операторы с более низким приоритетом, and и or, которые кому-то кажутся более удобочитаемыми и не требуют использования скобок в списочных операторах. Эти операторы также выполняют сокращенные вычисления. Полный список можно найти в табл. 3.8.

Оператор диапазона

Оператор диапазона .. представляет собой, в зависимости от контекста, два разных оператора.

В скалярном контексте .. возвращает логическое значение. Оператор диапазона, подобно электронному триггеру, имеет два устойчивых состояния и эмулирует оператор диапазона строк (запятая) из *sed*, *awk* и различных редакторов. Каждый скалярный оператор сохраняет собственное логическое состояние. Оно имеет значение «ложь», если левый операнд имеет значение «ложь». Если левый операнд имеет значение «истина», оператор диапазона остается истинным, пока правый операнд не получит значение «истина», *после* чего оператор диапазона снова получает значение «ложь». Оператор не получит значения «ложь», пока не будет вычислен в очередной раз. Он может проверить правый операнд и стать ложным в том же вычислении, когда стал истинным (так ведет себя оператор диапазона *awk*), но все равно один раз вернет значение «истина». Если вы не хотите, чтобы правый операнд проверялся до следующего вычисления (а так действует оператор диапазона *sed*), нужно вместо двух точек использовать три (...).[1] Оба оператора, и ., не проверяют правый операнд, находясь в состоянии «ложь», и не проверяют левый операнд, находясь в состоянии «истина».

---

[1] **Не путайте оператор диапазона ... с инструкцией ..., возбуждающей исключение "Unimplemented" (не реализовано).**

Возвращаемое значение является либо пустой строкой для значения «ложь», либо порядковым номером (начинающимся с 1) для значения «истина». Последовательный номер сбрасывается для каждого встретившегося диапазона. К последнему номеру в диапазоне добавляется строка "E0", которая не влияет на числовое значение, но дает значение, в котором можно осуществлять поиск при желании исключить конечную точку. Исключить начальную точку можно, дождавшись, когда последовательный номер станет больше 1. Если оба операнда скалярного .. представляют собой числовые литералы, то операнды неявно сравниваются с переменной $., которая содержит номер строки входного файла.[1]

Примеры:

```
if (101 .. 200) { print }   # вывести вторую сотню строк
next line if 1 .. /^$/,     # пропустить заголовочные строки сообщения
s/^/> / if /^$/ .. eof(),   # заключить в кавычки тело сообщения
```

В списочном контексте .. возвращает список значений от левого до правого с шагом в единицу. Это удобно при записи циклов for (1..10) и для выполнения операций над срезами массивов:

```
for (101 .. 200) { print } # выводит 101102...199200

my @foo = getlist();
@foo = @foo[0 .. $#foo];   # дорогостоящая пустая операция
@foo = @foo[ -5 .. -1];    # срез из последних 5 элементов
```

В текущей реализации, когда оператор диапазона используется в выражении цикла *foreach*, временный массив не создается, но в старых версиях Perl можно было «спалить» немало памяти, записав такой цикл:

```
for (1 .. 1_000_000) {
    # код
}
```

Если левое значение больше правого, возвращается пустой список. (Для создания списка с элементами в обратном порядке используется оператор reverse.)

Если операнды являются строками, оператор диапазона применяет алгоритм волшебного автоинкрементирования, обсуждавшийся ранее. Поэтому можно сказать:

```
my @alphabet = ("A" .. "Z");
```

и получить все буквы (английского) алфавита, либо:

```
my $hexdigit = (0 .. 9, "a" .. "f")[$num & 15];
```

и получить шестнадцатеричную цифру, либо:

```
my @z2 = ("01" .. "31");
print $z2[$mday];
```

и получить список дат с отбивкой нулями. Можно также сказать:

```
my @combos = ("aa" .. "zz");
```

---

[1] Точнее, она содержит количество вызовов оператора readline для данного дескриптора.

и получить все комбинации из двух букв нижнего регистра. Однако остерегайтесь таких выражений:

```
my @bigcombos = ("aaaaaaa" .. "zzzzzzz");
```

поскольку для них требуется большой объем памяти. Скажем точно, потребуется память для записи 8 031 810 176 скаляров. Остается надеяться, что в вашем распоряжении 64-разрядная машина. С терабайтом оперативной памяти. *Быстрой* памяти. Возможно, итеративный подход лучше.

Если значение второго операнда находится за пределами последовательности, которую может создать волшебный инкремент, перебор значений в последовательности продолжается, пока очередное значение не окажется длиннее значения в правом операнде. Например, выражение `"W" .. "M"` воспроизведет последовательность `"W"`, `"X"`, `"Y"` и `"Z"`, а затем остановится, потому что следующий элемент последовательности, `"AA"`, окажется длиннее `"M"`.

Если значение левого операнда не является частью последовательности, воспроизводимой волшебным инкрементом (т.е. непустая строка, соответствующая шаблону `/^[a-zA-Z]*[0-9]*\z/`), оператор вернет только начальное значение. Поэтому в следующем фрагменте будет получено только alpha:

```
use charnames "greek";
my @greek_small = ("\N{alpha}" .. "\N{omega}");
```

Чтобы получить буквы греческого алфавита в нижнем регистре, можно воспользоваться следующим приемом:

```
use charnames "greek";
my @greek_small = map { chr } (
                        ord("\N{alpha}") .. ord("\N{omega}")
                    );
```

Однако при этом будут отобраны лишние буквы, потому что между буквами «ро» и «тау» следуют две разные буквы «сигма» нижнего регистра – кроме основного имеется дополнительный символ `"\N{final sigma}"`. В общем случае числовой порядок следования кодов символов редко совпадает с алфавитным порядком следования символов. Подробности читайте в разделе «Сравнение и сортировка текста Юникода» в главе 6.

Условный оператор

Как и в языке C, `?:` является единственным тернарным оператором. Его часто называют условным оператором, поскольку он работает во многом так же, как конструкция `if-then-else`, за исключением того, что, поскольку это выражение, а не команда, его можно безопасно встраивать в другие выражения и вызовы функций. Так как это тернарный оператор, две его части разделяют три выражения:

```
COND ? THEN : ELSE
```

Если условие *COND* истинно, вычисляется только выражение *THEN*, значение которого становится значением выражения в целом. В противном случае вычисляется только выражение *ELSE*, значение которого становится значением выражения в целом.

Скалярный или списочный контекст распространяется далее на второй или третий аргумент в зависимости от того, который из них выбран. (Первый аргумент всегда находится в скалярном контексте, поскольку этот оператор является условным.)

```
my $a = $ok ? $b : $c; # получить скаляр
my @a = $ok ? @b : @c; # получить массив
my $a = $ok ? @b : @c; # получить счетчик числа элементов массива
```

Условный оператор часто внедряют в список значений для форматирования посредством `printf`, поскольку никому не хочется повторять целый оператор ради того, чтобы переключиться между двумя связанными значениями.

```
printf "I have %d camel%s.\n",
               $n,     $n == 1 ? "" : "s";
```

Удобно, что приоритет ?: выше, чем у запятой, но ниже, чем у большинства операторов, используемых внутри выражения (== в данном примере), поэтому обычно не приходится расставлять скобки. Но для ясности можно, при желании, добавить и скобки. Для условных операторов, вложенных в части *THEN* других условных операторов, мы предлагаем использование перевода строки и отступа, как если бы это были обычные операторы `if`:

```
$leapyear =
    $year % 4 == 0
        ? $year % 100 == 0
            ? $year % 400 == 0
                ? 1
                : 0
            : 1
        : 0;
```

Для условных операторов, вложенных в части *ELSE* предшествующих условных операторов, можно сделать аналогичную вещь:

```
$leapyear =
    $year % 4
        ? 0
        : $year % 100
            ? 1
            : $year % 400
                ? 0
                : 1;
```

но, как правило, лучше выстроить все части *COND* и *THEN* вертикально:

```
$leapyear =
    $year % 4 ? 0 :
    $year % 100 ? 1 :
    $year % 400 ? 0 : 1;
```

Выравнивание вопросительных знаков и двоеточий способно прояснить даже довольно запутанные конструкции:

```
printf "Да, мне нравится моя %s книга!\n",
    $i18n eq "french"   ? "chameau"
```

```
    $i18n eq "german"   ? "Kamel"
    $i18n eq "japanese" ? "\x{99F1}\x{99DD}" :
                          "camel"
```

При использовании прагмы `utf8` символы Юникода можно даже не экранировать:

```
use utf8;
printf "Да, мне нравится моя %s книга!\n",
    $i18n eq "french"   ? "chameau" :
    $i18n eq "german"   ? "Kamel"   :
    $i18n eq "japanese" ? "駱駝"    :
                          "camel"
```

Можно осуществлять присваивание условному оператору,[1] если второй и третий аргументы могут выступать в качестве l-значений (т.е. допустимо осуществлять присваивание им) и оба являются либо скалярами, либо списками (в противном случае Perl не будет знать, какой контекст предоставить правой части присваивания):

```
($a_or_b ? $a : $b) = $c; # либо $a, либо $b получает значение $c
```

Имейте в виду, что условный оператор связывает аргументы сильнее, чем различные операторы присваивания. Обычно это и требуется (взгляните, например, на вышеприведенные присваивания `$leapyear`), но изменить это без применения скобок нельзя. Присваивание без скобок может привести к неприятностям, и при этом может не возникнуть ошибка синтаксического анализа, поскольку анализатор Perl может счесть условный оператор за левое значение. Например, можно написать так:

```
$a % 2 ? $a += 10 : $a += 2    # НЕВЕРНО
```

Но это будет интерпретироваться так:

```
(($a % 2) ? ($a += 10) : $a) += 2
```

Операторы присваивания

Perl признает операторы присваивания C, добавляя к ним некоторые собственные. Их много:

```
=    **=    +=    *=    &=    <<=    &&=
            -=    /=    |=    >>=    ||=
            .=    %=    ^=
                  x=
```

Каждый такой оператор ожидает в качестве цели получить l-значение (обычно переменную или элемент массива) в левой части и выражение в правой части. Для простого оператора присваивания:

```
ЦЕЛЬ = ВЫРАЖЕНИЕ
```

[1] Удобочитаемость вашей программы это вряд ли повысит, но зато можно сделать весомую заявку на победу в конкурсе на самый непонятный (obfuscated) код на Perl.

значение *EXPR* запоминается в переменной или по адресу, указанному в *TARGET*. Для других операторов Perl вычисляет выражение:

```
ЦЕЛЬ ОП= ВЫРАЖЕНИЕ
```

как если бы оно было записано:

```
ЦЕЛЬ = ЦЕЛЬ ОП ВЫРАЖЕНИЕ
```

Это удобное умозрительное правило, но оно порождает некоторые заблуждения. Во-первых, все операторы присваивания всегда имеют приоритет обычного присваивания, независимо от приоритета, который операция *ОП* имела бы отдельно. Во-вторых, *ЦЕЛЬ* вычисляется только один раз. Обычно это не имеет значения, если нет побочных эффектов, таких как автоинкрементирование:

```
$var[$a++] += $value,             # $a инкрементируется один раз
$var[$a++] = $var[$a++] + $value;  # $a инкрементируется два раза
```

В отличие от присваивания в C, оператор присваивания в Perl возвращает работоспособное l-значение. Модификация присваивания аналогична выполнению присваивания с последующей модификацией переменной, которой было присвоено значение. Это удобно при модификации копии чего-либо, например:

```
($tmp = $global) += $constant;
```

что эквивалентно:

```
$tmp = $global + $constant;
```

Аналогично:

```
($a += 2) *= 3;
```

эквивалентно:

```
$a += 2;
$a *= 3;
```

Это не Бог весть как удобно, но вот идиома, которая встречается часто:

```
(my $new = $old) =~ s/foo/bar/g;
```

В Perl v5.14 и более поздних версий ее можно записывать таким же образом, но с применением модификатора /r, что приводит к возврату копии измененной версии переменной, а не переменной, с которой осуществляет связывание оператор =~.

```
my $new = ($old =~ s/foo/bar/gr);
my $new =  $old =~ s/foo/bar/gr·
```

Во всех случаях значением оператора присваивания является новое значение переменной. Поскольку операторы присваивания ассоциативны справа налево, это можно использовать для присваивания нескольким переменным одного и того же значения:

```
$a = $b = $c = 0;
```

При этом 0 присваивается $c, результат этого действия (по-прежнему 0) присваивается $b, а результат этого действия (*по-прежнему* 0) присваивается $a.

Присваивание списку может выполняться только простым оператором присваивания =. В списочном контексте присваивание списку возвращает список новых значений – так же, как это делает скалярное присваивание. В скалярном контексте присваивание списку возвращает число значений, имевшихся в правой части присваивания, как отмечалось в главе 2. Благодаря этому такое присваивание удобно использовать для проверки значений функций, возвращающих пустой список, когда их вызов оказывается безуспешным (или перестает быть успешным), например:

```
while (($key, $value) = each %gloss) { ... }

next unless ($dev, $ino, $mode) = stat $file;
```

Оператор запятой

Бинарный оператор, является оператором запятой. В скалярном контексте он вычисляет свой левый аргумент в пустом контексте, отбрасывает полученное значение, затем вычисляет свой правый аргумент в скалярном контексте и возвращает полученное значение. Это действие вполне идентично действию оператора запятой в С. Например:

```
$a = (1, 3);
```

присваивает $a значение 3. Не следует путать использование этого оператора в скалярном и списочном контекстах. В списочном контексте запятая служит просто разделителем аргументов и вставляет оба свои аргумента в *LIST*. Никакие значения она не отбрасывает.

Например, если изменить предыдущий пример следующим образом:

```
@a = (1, 3);
```

то будет создан список из двух элементов, а

```
atan2(1, 3),
```

является вызовом функции atan2 с двумя аргументами.

Диграф => служит в основном синонимом оператора запятой. Он удобен при записи аргументов, которые задаются парами. Кроме того, он заставляет интерпретировать любой идентификатор, находящийся непосредственно слева от него, как строку. Такое автоматическое цитирование работает только для идентификаторов и не работает для числовых литералов.

Списочные операторы (вправо)

Правая часть списочного оператора управляет всеми аргументами списочного оператора, которые разделяются запятыми, поэтому приоритет списочного оператора ниже, чем запятой, если смотреть в правую сторону. Когда списочный оператор начинает перемалывать аргументы, разделяемые запятыми, остановить его могут только лексемы, завершающие целое выражение (например, точка с запятой или модификатор команды), либо лексемы, завершающие текущее подвыражение

(например, правая круглая или квадратная скобка), либо логические операторы с низким приоритетом, о которых мы и поговорим в следующем разделе.

Логические and, or, not и xor

В качестве альтернативы операторам &&, || и ! Perl предоставляет операторы and, or и not, имеющие низкий приоритет. Принцип действия этих операторов попарно идентичен; в частности, and и or вычисляются по короткой схеме, как и их аналоги, что делает их полезными не только в логических выражениях, но и для управления логикой программы.

Поскольку приоритет этих операторов значительно ниже, чем у заимствованных из C, их можно безопасно использовать после списочных операторов без всяких скобок:

```
unlink "alpha", "beta", "gamma"
        or gripe(), next LINE;
```

Применяя операторы в стиле C, следовало бы написать нечто вроде:

```
unlink("alpha", "beta", "gamma")
        || (gripe(), next LINE);
```

Но нельзя просто взять и заменить все вхождения || на or. Допустим, что команду

```
$xyz = $x || $y || $z;
```

мы заменим такой:

```
$xyz = $x or $y or $z; # НЕВЕРНО
```

Она сделает совсем другое! Приоритет присваивания выше, чем приоритет or, но ниже, чем приоритет ||, поэтому сначала переменная $x будет присвоена переменной $xyz, а затем выполнены or. Чтобы получить такой же результат, как с использованием ||, нужно написать:

```
$xyz = ( $x or $y or $z );
```

Мораль истории такова: выучить приоритеты (или использовать скобки) придется вне зависимости от того, с какими логическими операторами вы работаете. Мы предлагаем использовать круглые скобки в любых конструкциях, которые могут вызывать сомнения у других, даже если они не вызывают сомнений у вас.[1]

Существует также логический оператор xor, не имеющий точного аналога в C или Perl, поскольку единственный предоставляемый ими оператор исключающего ИЛИ (^) действует поразрядно. Оператор xor не может вычисляться по короткой схеме, поскольку всегда необходимо вычислить операнды с обеих сторон. Лучшим эквивалентом для `$a xor $b` является, вероятно, `!$a != !$b`. Конечно, можно написать `!$a ^ !$b` или даже `$a ? !$b : !!$b`. Суть в том, что $a и $b должны быть вычислены как true или false в логическом контексте, а имеющийся поразрядный оператор не предоставляет его без дополнительных усилий.

---

[1] Если только вы не собираетесь *заставить* других выучить приоритеты операторов, в чем мы можем вас только поддержать.

Операторы C, отсутствующие в Perl

Вот что есть в C и чего нет в Perl:

унарный &

Оператор взятия адреса. Однако оператор \ в Perl (для получения ссылки) занимает ту же экологическую нишу:

```
$ref_to_var = \$var;
```

Но ссылки Perl значительно безопаснее, чем указатели C.

унарный *

Оператор разыменования адреса. Поскольку в Perl нет адресов, нет нужды и в их разыменовании. Однако ссылки в нем есть, поэтому символы префиксов переменных Perl служат как операторы разыменования, а также указывают на тип: $, @, % и &. Несколько странно, что фактически имеется оператор разыменования *, но поскольку * является разыменовывающим префиксом, обозначающим typeglob, он используется иначе.

(*ТИП*)

Оператор приведения типа. Какому типу захочется, чтобы его приводили?

4

Операторы и объявления

Программа на Perl состоит из последовательности объявлений (declarations) и операторов (statements). Объявление может находиться в любом месте, где допускается оператор, но основную роль оно играет на этапе компиляции программы. Некоторые объявления несут двойную нагрузку и выполняют также роль обычных операторов, но в большинстве своем совершенно прозрачны на этапе выполнения. После компиляции главная последовательность операторов выполняется только один раз.

В этой главе объявления рассматриваются после операторов, так что нам хотелось бы упомянуть о наиболее важных объявлениях прямо сейчас.

В отличие от большинства языков программирования, по умолчанию Perl не требует явного объявления переменных. Они начинают существовать при первом обращении к ним, независимо от того, были они объявлены или нет. Но при желании можно объявить переменные заранее, с помощью *объявлений* my, our и state, предшествующих именам переменных, благодаря чему компилятор, встретив позже имя этой переменной, будет уверен, что оно не содержит опечаток.

По умолчанию, при попытке обратиться к переменной, которой ни разу не присваивалось значение, считается, что переменная хранит нулевое значение (если используется как число) или "" – пустую строку (если используется как строка), или же получает значение «ложь» при обращении к ней как к логической переменной. Если программист хочет получать предупреждения об использовании неопределенных значений, как если бы они действительно были строками или числами, или даже рассматривать такие действия как ошибку, об этом позаботится объявление use warnings.

Аналогично с помощью объявления use strict можно потребовать от самого себя объявлять все переменные заранее. В этом случае обращение к необъявленным переменным будет считаться синтаксической ошибкой. (Чтобы включить строгий режим, достаточно также добавить объявление use v5.12 или выше, но мы рекомендуем use v5.14, чтобы все примеры в этой книге компилировались и выполнялись без проблем.) Подробнее об этих объявлениях рассказывается ближе к концу этой главы, в разделе «Прагмы».

Простые операторы

Простой оператор – это выражение, вычисляемое ради его побочных эффектов. Каждый простой оператор должен заканчиваться точкой с запятой, если он не является последним оператором в блоке. В последнем случае точка с запятой не обязательна – Perl понимает, что оператор завершен, поскольку закончился блок. Однако в конце блока из нескольких строк точку с запятой лучше поставить, поскольку потом, возможно, будет добавлена еще одна строка.

Операторы вроде eval {}, do {} и sub {} выглядят как составные операторы, но в действительности таковыми не являются. Верно, внутри них допускается наличие нескольких операторов, но это не в счет. Внешне эти операторы являются просто термами в выражении, и потому требуют явного использования точки с запятой, если используются как последний элемент оператора.

За любым простым оператором может следовать один необязательный модификатор, как раз перед завершающей точкой с запятой (или окончанием блока). Допускаются следующие модификаторы:

```
if EXPR
unless EXPR
while EXPR
until EXPR
foreach LIST
```

Модификаторы if (если) и unless (если только не) действуют весьма схожим с обычным языком образом:

```
$trash->take("out") if $you_love_me;
shutup() unless $you_want_me_to_leave;
```

Модификаторы while и until вычисляются многократно. Модификатор while обеспечивает повторное выполнение, пока его выражение остается истинным, а модификатор until обеспечивает повторное выполнение, пока его выражение остается ложным:

```
$expression++ while -e "$file$expression
kiss('me') until $I_die;
```

Модификатор for (можете также использовать вариант написания foreach, если вам не жалко свою клавиатуру) выполняет одно вычисление для каждого элемента в его списке *LIST*, при этом $_ служит псевдонимом текущего элемента:

```
s/java/perl/ for @resumes;
say "поле: $_" foreach split /:/  $dataline;
```

Модификаторы while и until имеют обычную семантику цикла while (сначала вычисляется условие), за исключением применения к блоку do (глава 27), когда блок один раз выполняется перед тем, как вычисляется условие. Благодаря этому можно писать такие циклы:

```
do {
    $line = <STDIN>;
    ...
} until $line eq ".\n";
```

Обратите также внимание, что операции управления циклом, описываемые ниже, в этой конструкции не будут работать, поскольку модификаторы не принимают метки циклов. Всегда можно заключить конструкцию в дополнительный блок для досрочного завершения или же поместить дополнительный блок внутри нее для досрочной итерации, как описано ниже, в разделе «Голые блоки в роли циклов». Либо сконструировать настоящий цикл, содержащий несколько операций управления.

Модификатор when – экспериментальная функция, доступная только при наличии объявления use v5.14 (или выше). Семантика сопоставления этого модификатора совпадает с семантикой оператора when, поэтому за дополнительной информацией обращайтесь к разделу «Оператор и модификатор when» ниже в этой главе.

Составные операторы

Последовательность операторов в определенной области видимости[1] называется *блоком*. Иногда областью видимости является весь файл, например, файл, загружаемый директивой require, или файл, содержащий основной текст программы. Иногда областью видимости является строка, вычисляемая через eval. Но обычно блок – это то, что заключено в фигурные скобки ({}). Под областью видимости подразумевается любой из этих трех вариантов. Когда же речь идет о блоке в фигурных скобках, применяется термин «блок» (*BLOCK*).

Составные операторы образуются из выражений и блоков (*BLOCK*). Выражения состоят из термов и операторов. В наших синтаксических описаниях словом *EXPR* мы будем обозначать место, где можно использовать любое скалярное выражение. Для обозначения выражения, вычисляемого в списочном контексте, будем говорить *LIST*.

Следующие конструкции можно применять для управления условным и многократным выполнением блоков *BLOCK*. (Часть *LABEL* является необязательной.)

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

unless (EXPR) BLOCK
unless (EXPR) BLOCK else BLOCK
unless (EXPR) BLOCK elsif (EXPR) BLOCK ...
unless (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

given (EXPR) BLOCK

LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK

LABEL until (EXPR) BLOCK
LABEL until (EXPR) BLOCK continue BLOCK

LABEL for (EXPR; EXPR; EXPR) BLOCK

LABEL foreach (LIST) BLOCK
```

---

[1] Области видимости и пространства имен описаны в разделе «Имена» главы 2.

```
LABEL foreach VAR (LIST) BLOCK
LABEL foreach VAR (LIST) BLOCK continue BLOCK

LABEL BLOCK
LABEL BLOCK continue BLOCK
```

Обратите внимание на то, что, в отличие от C и Java, эти конструкции определены в терминах блоков, а не операторов. Это означает, что фигурные скобки необходимы – присутствие автономных операторов не допускается. Условный оператор без фигурных скобок можно записать несколькими способами. Все следующие команды делают одно и то же:

```
unless (open(FOO, '<', $foo)) { die "Невозможно открыть $foo: $!" }
if (!open(FOO, '<', $foo))    { die "Невозможно открыть $foo: $!" }

die "Невозможно открыть $foo: $!"   unless open(FOO, '<', $foo);
die "Невозможно открыть $foo: $!"   if !open(FOO, '<', $foo);

open(FOO, '<', $foo)   || die "Невозможно открыть $foo: $!";
open(FOO, '<', $foo)   or die "Невозможно открыть $foo: $!";
```

В большинстве случаев мы предпочитаем последнюю пару. Они визуально не столь перегружены, как остальные, особенно вариант `or die`. В форме с оператором `||` приходится дисциплинированно расставлять скобки, но в версии `or` о них можно и позабыть.

Но главное, почему нам более симпатичны последние два варианта: в них самая важная часть оператора располагается в начале строки – на виду у читателя. Обработка ошибки отходит в сторону, так что можно не обращать на нее внимания, пока в этом не возникнет необходимость.[1] Если каждый раз табулировать все проверки `or die` таким образом, чтобы они начинались в одной и той же колонке справа, чтение станет еще более удобным:

```
chdir $dir             or die "Смена каталога $dir: $!";
open(FOO, '<', $file)  or die "Открытие $file: $!";
@lines = <FOO>         or die "$file пуст?";
close(FOO)             or die "Закрытие $file: $!";
```

Операторы if и unless

Оператор `if` прост. Поскольку блоки всегда заключены в фигурные скобки, неоднозначности в отношении того, к которому `if` относится тот или иной оператор `else` или `elsif`, не возникает. Из каждой данной последовательности блоков `if`/`elsif`/`else` выполняется только тот первый, значение условия которого истинно. Если ни одно условие не является истинным, выполняется блок `else`, если он имеется. Обычно полезно поместить `else` в конце цепочки `elsif` на случай, если вы забыли обработать один из вариантов.

Если вместо `if` используется `unless`, смысл проверки условия становится противоположным. Таким образом:

```
unless ($x == 1)
```

равносильно:

[1] (Как в этой сноске.)

```
if ($x != 1)
```

или даже неприглядному:

```
if (!($x == 1)) .
```

Область видимости переменной, объявленной в управляющем условии, простирается от места ее объявления и до конца условного оператора, включая все блоки elsif **и завершающую ветвь** else**, если они имеются, но не далее того:**

```
if ((my $color = <STDIN>) =~ /red/i) {
    $value = 0xff0000;
}
elsif ($color =~ /green/i) {
    $value = 0x00ff00;
}
elsif ($color =~ /blue/i) {
    $value = 0x0000ff;
}
else {
    chomp $color;
    warn "неизвестная составляющая RGB `$color', использован черный\n"
    $value = 0x000000;
}
```

По завершении блока else **переменная** $color **больше не видима. Если нужно увеличить область видимости, объявите переменную до условного оператора.**

Оператор given

В предыдущем примере речь шла о цвете. Лингвисты называют это *темой*. В Perl v5.10 появился оператор given**, представляющий альтернативу конструкциям из операторов** if**. Если говорить языком лингвистики, оператор** given ***определяет тему*****, устанавливая ее в переменной** $_**. После этого можно использовать операторы** when **для проверки соответствия темы различным значениям или шаблонам.**

Эта особенность доступна при использовании версии Perl не ниже v5.10:

```
use v5.12;              # не ниже v5.12, загрузить особенности по умолчанию
```

А также, если явно включена поддержка «switch»:

```
use feature "switch"  # включить поддержку switch
```

Любая из этих инструкций добавляет в Perl ряд новых ключевых слов: given**,** when**,** break**,** continue **и** default**. Ниже представлен один из способов реализации предыдущего примера с применением новых возможностей:**

```
use v5.10;

my $value;
given (<STDIN>) {
    when (/red/i)   { $value = 0xFF0000 }
    when (/green/i) { $value = 0x00FF00 }
    when (/blue/i)  { $value = 0x0000FF }
    default {
        chomp;
```

```
        warn "неизвестная составляющая RGB `$_`, использован черный\n";
        $value = 0x000000;
    }
}
```

Версия v5.10, по сути, *вынуждала* написать именно так, потому что given в этой версии не позволяет возвращать значения. Начиная с версии v5.14 появилась возможность возвращать значения, и теперь, применив модификаторы оператора when, наш пример можно записать так:

```
use v5.14;

my $value = do {
    given (<STDIN>) {
        0xFF0000 when /red/i;
        0x00FF00 when /green/i;
        0x0000FF when /blue/i;
        chomp;
        warn "неизвестная составляющая RGB `$_` использован черный\n";
        0x000000;
    }
};
```

Аргументы передаются операторам given и when в скалярном контексте. Оператор given связывает свой аргумент с переменной $_, устанавливая тему для своего блока. Оператор when на основе типа своего аргумента определяет разновидность сопоставления с шаблоном. Семантика оператора when является надмножеством семантики интеллектуального сопоставления. Если аргумент выглядит, как логическое выражение, он вычисляется непосредственно. В противном случае он передается оператору интеллектуального сопоставления для последующей интерпретации в виде $_ ~~ *EXPR*. Это может показаться сложным, но в действительности это не так, потому что в большинстве своем инструкции выбора имеют вид:

```
use v5.14;

my $n = somefunc();

given ($n) {
    when (0)        { say "ноль" }
    when (1)        { say "один" }
    when ([3..7])   { say "несколько" }
    when (/^\d+$/)  { say "много" }
    default         { say "непонятно" }
}
```

Иными словами, операторам when обычно передаются аргументы, вызывающие интеллектуальное сопоставление.

Ниже приводится более длинный пример оператора given:

```
use feature ":5.10";

given ($n) {

# соответствует: if !defined($n)
    when (undef) {
        say '$n имеет неопределенное значение';
```

```
    }

# соответствует: if $n eq "foo"
    when ("foo") {
        say "$n - строка "foo"";
    }

# соответствует: if $n ~~ [1,3,5,7,9]
    when ([1,3,5,7,9]) {
        say "$n - нечетная цифра";
        continue; # продолжить сопоставление!!
    }

# соответствует: if $n < 100
    when ($_ < 100) {
        say "$n меньше 100 в числовом контексте";
    }

# соответствует: if complicated_check($n)
    when (\&complicated_check) {
        say "сложная проверка $n дает истину";
    }

# когда не соответствует ни одному другому случаю
    default {
        die q(Невозможно определить, что делать с $n);
    }
}
```

Оператор given(*EXPR*) присваивает значение выражения *EXPR* копии $_ в лексической области видимости, а не псевдониму в динамической области, как это делает оператор foreach в отсутствие объявления my. Это делает его похожим на блок do:

```
do { my $_ = EXPR; ... }
```

за исключением того, что в случае успеха when (или явный оператор break) знает, как выйти за пределы блока. Поскольку оператор given образует лексическую область видимости, его нельзя использовать для локализации динамического значения $_, как это позволяет делать foreach старого образца.[1]

Ключевое слово break можно использовать для выхода за пределы охватывающего блока given. Каждый блок when неявно завершается ключевым словом break.

Выйти из блока when и перейти к началу следующей инструкции, которая может быть, а может не быть другим оператором when, можно с помощью ключевого слова continue:

```
given($foo) {
    when (/x/) { say "$foo содержит x"; continue }
    say "Эта строка выводится всегда.";
    when (/y/) { say "$foo содержит y" }
    default    { say "$foo не содержит y" }
}
```

[1] Этот недостаток скорее следует рассматривать как особенность, так как использование $_ с динамической областью видимости чревато ошибками, поскольку в этом случае мы получаем два различных фрагмента кода, конкурирующих за текущую тему.

Когда оператор given является также допустимым выражением (например, когда он является последней инструкцией в блоке), он возвращает:

- **пустой список, как только в коде встретился явный оператор break;**
- **значение последнего вычисленного выражения выполнившейся ветви when/ default, если таковая имеется;**
- **значение последнего вычисленного выражения в блоке given, если ни одно из соответствий не сработало.**

Последнее выражение вычисляется в контексте блока given в целом. Обратите внимание, что, в отличие от операторов if и unless, оператор when с ложным условием всегда возвращает пустой список.

```
my $price = do {
    given ($item) {
        when (["груша", "яблоко"]) { 1 }
        break when "vote"; # Мой голос не продается
        1e10 when /Мона Лиза/;
        "неизвестно";
    }
};
```

Обратите внимание, что в примере выше пришлось использовать блок do, иначе оператор given интерпретируется как обычная инструкция, и не может располагаться справа от оператора присваивания. (Возможно, в будущих версиях Perl мы сделаем использование do необязательным.)

Оператор и модификатор when

Мощь оператора given сосредоточена, по преимуществу, в неявном интеллектуальном сопоставлении, которое подразумевается различными типами данных. Конструкция when(*EXPR*) по умолчанию интерпретируется как неявное интеллектуальное сопоставление переменной $_ с выражением *EXPR*; т.е. как $_ ~~ *EXPR*. (Подробнее об интеллектуальном сопоставлении рассказывается в главе 3.) Однако, если аргумент *EXPR* оператора when имеет одну из перечисленных ниже специальных форм, результат вычисления этого выражения *EXPR* преобразуется в логическое значение, а интеллектуальное сопоставление не выполняется:

1. **Вызов пользовательской подпрограммы или метода.**
2. **Сопоставление с регулярным выражением в форме /*REGEX*/, $foo =~ /*REGEX*/ или $foo =~ *EXPR*.**
3. **Выражение, явно использующее ~~ (оператор интеллектуального сопоставления), такое как *EXPR* ~~ *EXPR*. (Явное интеллектуальное сопоставление переменной $_ может потребоваться, например, чтобы обратить использование встроенного полиморфизма механизма интеллектуального сопоставления оператора when.)**
4. **Оператор отношения, такой как $_ < 10 или $x eq "abc", возвращающий логический результат. Сюда относятся шесть операторов сравнения чисел (<, >, <=, >=, == и !=) и шесть операторов сравнения строк (lt, gt, le, ge, eq и ne).**
5. **Три встроенные функции: defined, exists и eof.**
6. **Операторы отрицания выражений !*EXPR* и not(*EXPR*), а также логическое ИСКЛЮЧАЮЩЕЕ ИЛИ, *EXPR1* xor *EXPR2*. (Поразрядные версии этих операторов,**

- и ~, сюда не относятся.) В эту категорию также попадает отрицание регулярного выражения в любой форме записи !/REGEX/, $foo !~ /REGEX/ или $foo !~ EXPR.

7. Операторы проверки файлов (кроме -s, -M, -A и -C, так как они возвращают числовые значения, а не логические).
8. Операторы и .. (Обратите внимание, что инфиксная операция совершенно отличается от многоточия ..., которое распознается только там, где ожидается оператор.)

В первых восьми случаях операнд *EXPR* используется непосредственно как логическое значение, поэтому интеллектуальное сопоставление не выполняется. Оператор when можно считать *еще более интеллектуальным* оператором интеллектуального сопоставления.[1] Чтобы сделать его *еще более интеллектуальным*, Perl применяет все эти проверки к операндам логических операторов (т.е. «И» и «ИЛИ») рекурсивно, чтобы определить, следует ли использовать интеллектуальное сопоставление, как описывается ниже:

1. В выражениях *EXPR1* && *EXPR2* и *EXPR1* and *EXPR2* проверка применяется рекурсивно к обоим операндам, *EXPR1* и *EXPR2*. Все выражение считается логическим, только если оба операнда прошли проверку. В противном случае выполняется интеллектуальное сопоставление.
2. В выражениях *EXPR1* || *EXPR2* и *EXPR1* or *EXPR2* проверка рекурсивно применяется только к операнду *EXPR1* (который, например, сам может быть логическим выражением с высокоприоритетным оператором И, и потому подпадать под действие предыдущего правила), но не к операнду *EXPR2*. Если операнд *EXPR1* требует применения интеллектуального сопоставления, ко второму операнду *EXPR2* также будет применено интеллектуальное сопоставление, независимо от типа операнда *EXPR2*. Но если первый аргумент в *EXPR1* не требует интеллектуального сопоставления, оно не будет использоваться и для второго аргумента. Этим данный тип выражений существенно отличается от выражений с оператором &&, описанных выше, поэтому будьте внимательны. (Имейте в виду, что выражения *EXPR1* // *EXPR2* всегда интерпретируются как логические, потому что подразумевают наличие функции defined слева от оператора //.)

Из-за всех этих правил жизнь начинает казаться сложнее, чем есть. Они необходимы потому, что в Perl 5 отсутствует встроенный логический тип.[2] Их назначение – выполнять те действия, которые вы подразумеваете. Например:

```
when (/^\d+$/ && $_ < 75) {     }
```

будет интерпретироваться как логическое выражение, потому что, согласно правилам, обе стороны выражения являются логическими.

А для:

```
when ([qw(foo bar)] && /baz/) {     }
```

---

[1] Полезно также считать логические выражения одной из разновидностей интеллектуального сопоставления, потому что в Perl 6 они таковыми и являются, и вам, возможно, время от времени придется менять свой взгляд на них.

[2] Пока, по крайней мере. Встроенный логический тип может появиться в будущих версиях Perl, и тогда все эти сложные правила отпадут естественным образом.

будет использовано интеллектуальное сопоставление, потому что только второй операнд выглядит как логическое значение. Первый операнд не похож на логическое значение, поэтому побеждает интеллектуальное сопоставление – иначе возникает логическая ошибка, если только не предполагается выполнить интеллектуальное сопоставление с *результатом* справа, имеющим значение 1 или "". Однако вы едва ли будете использовать для этого оператор given.

Помните, что в операциях дизъюнкции порядок имеет значение. Если сказать:

```
when ([qw(foo bar)] || /^baz/) { ... }
```

исходя из первого операнда, Perl применит интеллектуальное сопоставление. Но в случае

```
when (/^baz/ || [qw(foo bar)]) { ... }
```

слева находится регулярное выражение (логический операнд), из-за чего оба операнда будут считаться логическими. И снова возникает логическая ошибка, потому что второй аргумент (ссылка на массив) всегда принимает истинное значение, поэтому проверяться будет совсем не то, что предполагается.

Логические операторы, выполняющие операции с константами, подвергаются глубокой оптимизации. Бессмысленно писать:

```
when ("foo" or "bar") { ... }
```

Потому что это выражение будет оптимизировано до "foo", и значение "bar" никогда не будет рассматриваться (даже при том, что правила требуют применения интеллектуального сопоставления с "foo"). В случаях, когда требуется организовать сопоставление с несколькими альтернативами, как в данном примере, следует использовать ссылки на массивы, потому что это приведет к необходимости выполнить интеллектуальное сопоставление, имеющее собственную семантику «соответствует какому-либо элементу в»:

```
when (['foo', 'bar']) { ... }
```

Этот же прием используется для сопоставления нескольких «меток» с одним блоком логики, поскольку в Perl отсутствует семантика автоматического перехода к проверке последующих «меток», существующая в языке C.

Ветвь default действует в точности, как when(1 == 1), т.е. соответствует всегда. Поскольку предложения when вычисляются по порядку, предложение default должно быть последним – подобно when оно неявно выполняет оператор break, поэтому нижележащий код в операторе given не выполняется никогда.

В помощь семантике интеллектуального сопоставления, если в качестве аргумента оператору given передается литеральный массив или хеш, он будет преобразован в ссылку, чтобы не потерять какую-либо информацию. Поэтому, например, given(@foo) суть то же самое, что и given(\@foo). Если потребуется, чтобы в сопоставлении использовалась длина массива @foo, следует явно сказать given(scalar @foo).

Мы продолжаем считать некоторые особенности операторов given и when экспериментальными, но вы можете быть уверены, что инструкции выбора, основанные на сопоставлении с простыми строками и числами, всегда будут действовать, как вы того ожидаете.

Операторы циклов

Формальный синтаксис любого оператора цикла допускает использование необязательной метки *LABEL*. (Метку можно прикрепить к любому оператору, но в цикле она имеет особое значение.[1]) Метка состоит из идентификатора и следующего за ним двоеточия. Принято записывать метки символами в верхнем регистре, чтобы избежать конфликта с зарезервированными словами и для удобства чтения. (Perl не сбить с толку меткой, уже имеющей значение, например, меткой `if` или `open`, чего нельзя сказать о тех, кто будет читать вашу программу.)

Операторы while и until

Оператор `while` повторяет выполнение блока, пока выражение *EXPR* остается истинным. Если слово `while` заменить словом `until`, то смысл проверки становится противоположным, т.е. блок выполняется, только пока *EXPR* имеет значение «ложь». Однако условие все равно проверяется перед первой итерацией цикла.

Операторы `while` или `until` могут иметь необязательный дополнительный блок `continue`, который выполняется в начале итерации – при выходе за конец основного блока или при явном вызове `next` (операция управления циклом, которая вызывает переход к новой итерации). На практике блок `continue` применяется не слишком часто, но, рассказав о нем, мы теперь сможем строго определить трехчастный цикл `for` в следующем разделе.

В отличие от цикла `foreach`, который мы рассмотрим несколько позже, цикл `while` официально не имеет переменной цикла.[2] Однако у вас есть возможность объявить переменные явно. Область видимости переменных, объявленных в условном выражении оператора `while` или `until`, ограничивается одним или несколькими блоками, на которые распространяется действие этого условия. Они недоступны в охватывающей области видимости. Например:

```
while (my $line = <STDIN>) {
    $line = lc $line;
}
continue {
    print $line; # переменная пока видима
}
# теперь $line ушла из области видимости
```

В данном случае область видимости `$line` простирается от ее объявления в управляющем выражении на всю конструкцию цикла, включая блок `continue`, но не далее того. Если необходимо расширить область видимости, объявите переменную перед циклом.

---

1 До версии v5.14 метку нельзя было прикрепить к оператору `package`.

2 Как следствие, `while` не локализует переменные в проверяемом условии неявным образом. В результате могут возникнуть «интересные» последствия, если в цикле `while` используются операторы, неявно знающие о существовании таких глобальных переменных, как `$_`. В частности, в разделе «Оператор ввода строки (угловых скобок)» главы 2 вы можете узнать, как в некоторых циклах `while` может происходить неявное присваивание глобальной переменной `$_`, а также увидеть пример решения этой проблемы.

Трехчастные циклы

Круглые скобки трехчастного цикла[1] содержат три выражения, разделенные точкой с запятой. Эти выражения служат, соответственно, для инициализации, проверки условия и реинициализации цикла. Круглые скобки и две точки с запятой являются обязательными элементами, но сами выражения можно опустить. Выражения инициализации и реиницилизации, если они опущены, ничего не делают. Выражение проверки условия, если оно опущено, всегда считается истинным. (Значения выражений инициализации и реинициализации не играют никакой роли, поскольку эти выражения вычисляются только ради их побочных эффектов.)

Таким образом, трехчастный цикл можно определить через соответствующий цикл `while`, внедрив в него все три выражения. Когда вы говорите:

```
LABEL:
    for (my $i = 1; $i <= 10; $i++) {
        ...
    }
```

Perl за кулисами превращает ее в конструкцию, действующую так:

```
{
    my $i = 1;
LABEL:
    while ($i <= 10) {
        ...
    }
    continue {
        $i++;
    }
}
```

(за исключением того, что охватывающего блока на самом деле нет, и мы обозначили его, только чтобы показать границы области видимости [illegible]).

Если необходимо производить итерацию одновременно по двум переменным, просто разделите параллельные выражения запятыми:

```
my $i,
my $bit;
for ($i = 0, $bit = 0; $i < 32; $i++, $bit <<= 1) {
    say "Бит $i установлен" if $mask & $bit;
}
# значения $i и $bit сохраняются после конца цикла
```

Можно объявить эти переменные и так, чтобы они были видимы только внутри цикла:

```
for (my ($i, $bit) = (0, 1); $i < 32; $i++, $bit <<= 1) {
    say "Бит $i установлен" if $mask & $bit;
}
# теперь $i и $bit, объявленные для цикла, вышли из области видимости
```

---

[1] Также известен как цикл `for`, но это название может вас запутать, поскольку в Perl имеются циклы `for`, не являющиеся трехчастными. Из этих соображений мы стараемся неиспользовать данное название.

Помимо обычного перебора индексов массива, трехчастный цикл может служить многим другим интересных задачам. Ему даже не требуется явная переменная цикла. Вот пример того, как можно избежать проблемы, возникающей при явной проверке конца файла в интерактивном дескрипторе файла и приводящей программу к зависанию.

```
$on_a_tty = -t STDIN && -t STDOUT;
sub prompt { print "да? " if $on_a_tty }
for ( prompt(); <STDIN>; prompt() ) {
    # какие-то действия
}
```

Еще одно традиционное применение трехчастного цикла – «бесконечный цикл». Поскольку все три выражения не обязательны, а условие по умолчанию является истинным, цикл можно записать так:

```
for (;;) {
    ...
}
```

Это то же самое, что

```
while (1) {
    ...
}
```

Если вас смущает идея бесконечного цикла, то следует заметить, что всегда можно выйти из цикла в любой его точке, применив явную операцию управления циклом, такую как `last`. Конечно, если вы пишете программа управления для ядерной крылатой ракеты, явный выход из цикла может не потребоваться. В должное время цикл завершится автоматически.[1]

Циклы foreach

Эта разновидность циклов осуществляет перебор списка значений, последовательно устанавливая управляющую переменную (*VAR*) равной каждому элементу списка:

```
for my VAR (LIST) {
    ...
}
```

Если часть `my` *VAR* отсутствует, используется глобальная переменная `$_`. Определение `my` можно опустить, но только в отсутствие объявления `use strict`.

По историческим причинам синонимом ключевого слова `for` служит ключевое слово `foreach`, поэтому `for` и `foreach` можно использовать взаимозаменяемо и в зависимости от того, которое из них в конкретной ситуации лучше смотрится. Мы предпочитаем использовать `for`, потому что мы ленивы и потому что это слово удобнее читать, особенно с объявлением `my`. (Не беспокойтесь, Perl легко отличит `for (@ARGV)` от `for ($i=0; $i<$#ARGV; $i++)`, поскольку в последнем содержатся точки с запятой.) Вот несколько примеров:

---

1 То есть имеется тенденция к автоматическому падению из цикла.

```
$sum = 0;

for my $value (@array) { $sum += $value }

for my $count (10,9,8,7,6,5,4,3,2,1,'БУМ!') { # обратный отсчет
    say $count;
    sleep(1);
}

for (reverse 'БУМ!', 1 .. 10) {              # то же самое
    say;
    sleep(1);
}

for my $field (split /:/, $data) {           # любое списочное выражение
    say "Поле содержит: '$field'";
}

for my $key (sort keys %hash) {
    say "$key => $hash{$key}";
}
```

Последний пример представляет собой канонический способ вывода значений хеша в отсортированном порядке. Более подробные примеры можно найти в описаниях keys и sort в главе 27.

В процессе выполнения цикла for невозможно узнать, в какой точке списка вы находитесь. Можно запоминать предыдущий элемент списка в переменной и сравнивать соседние элементы, но часто приходится просто взять и написать трехчастный цикл for, перебирающий индексы. В конце концов, для того и существуют два типа цикла.

Если список *LIST* состоит только из значений, допускающих присваивание (обычно речь о переменных, а не о перечислениях констант), все эти переменные могут быть модифицированы путем изменения *VAR* внутри цикла. Это возможно потому, что переменная цикла представляет собой неявный псевдоним для каждого элемента обрабатываемого списка. Можно изменить не только один массив, а сразу несколько массивов и хешей, находящихся в одном списке:

```
for my $pay (@salaries) {            # всем прибавка 8%
    $pay *= 1.08;
}

for (@christmas, @easter) {          # изменить меню
    s/ham/turkey/;
}
s/ham/turkey/ for @christmas, @easter; # то же

for ($scalar, @array, values %hash) {
    s/^\s+//;                        # удалить начальные пробелы
    s/\s+$//;                        # удалить замыкающие пробелы
}
```

Переменная цикла действует только в динамической или лексической области видимости цикла и неявно становится лексической, если предварительно объявлена с помощью my. Благодаря этому она невидима для всех функций, определен-

ных вне области лексической видимости этой переменной, даже если они вызываются из этого цикла. Однако если в области видимости нет лексического объявления, переменная цикла станет локализованной глобальной переменной (с динамической областью видимости); это позволит функциям, вызываемым из цикла, обращаться к этой переменной. В любом случае, предыдущее значение, которое локализованная переменная имела до выполнения цикла, будет восстановлено при выходе из него.

При желании можно явно определить вид (лексический или глобальный) используемой переменной. Это облегчит работу тех, кто будет сопровождать ваш код; в противном случае им придется искать в охватывающих областях видимости предыдущее объявление, чтобы определить вид этой переменной:

```
for my $i    (1 .. 10) { ... } # $i всегда лексическая
for our $Tick (1 .. 10) { ... } # $Tick всегда глобальная
```

Если переменная цикла сопровождается объявлением, то предпочтительнее использовать более краткую форму for, а не foreach, поскольку она лучше читается на английском языке.

Вот как программист на C или Java мог бы поначалу решить запрограммировать некоторый алгоритм на Perl:

```
for ($i = 0; $i < @ary1; $i++) {
    for ($j = 0; $j < @ary2; $j++) {
        if ($ary1[$i] > $ary2[$j]) {
            last;              # Нельзя выйти во внешний цикл. :-(
        }
        $ary1[$i] += $ary2[$j];
    }
    # сюда приведет last
}
```

А вот как поступил бы ветеран программирования на Perl:

```
WID: for my $this (@ary1) {
    JET: for my $that (@ary2) {
        next WID if $this > $that;
        $this += $that;
    }
}
```

Видите, насколько легче это делается с помощью идиом Perl? Аккуратнее, надежнее и быстрее. Аккуратнее, поскольку меньше шума. Надежнее, поскольку если позже добавить код между внутренним и внешним циклами, новый код не будет выполнен по ошибке, так как операция next (о которой говорится ниже) явно выполняет переход на следующую итерацию внешнего цикла, а не просто выходит из внутреннего. И это быстрее, чем эквивалентный трехчастный цикл, поскольку обращение к элементам выполняется напрямую, а не через индексы.

Однако пишите так, как вам больше нравится. TMTOWTDI.

Подобно while, оператор foreach может иметь блок continue. Это позволяет выполнять фрагмент кода в конце каждой итерации цикла вне зависимости от того, попадаете вы в этот фрагмент выходом из итерации или посредством next.

И теперь мы можем наконец сказать: next на очереди.

Управление циклом

Мы уже говорили, что можно добавить метку в заголовок цикла, и тем самым дать ему имя. Метка цикла идентифицирует его для операций управления циклом, таких как next, last и redo. Метка обозначает цикл в целом, не только его первую строку. Поэтому операция управления циклом, ссылаясь на цикл, вовсе не осуществляет переход «go to» на метку цикла. Компьютеру все равно, и с таким же успехом метку можно было бы поместить в конце цикла, но по каким-то причинам люди предпочитают, чтобы метка стояла в начале.

Обычно циклам дают имя по тому элементу, который цикл обрабатывает в каждой итерации. Это прекрасно сочетается с операциями управления циклом, которые проектировались так, чтобы их можно было читать как обычный английский язык при использовании надлежащих меток и модификаторов операторов. Исконный цикл работает со строками, поэтому исконной меткой цикла является LINE:, а исконной операцией управления циклом нечто вроде:

```
next LINE if /^#/;      # отбросить комментарии
```

Синтаксис операций управления циклом следующий:

```
last LABEL
next LABEL
redo LABEL
```

Метка *LABEL* является необязательной; если она опущена, то оператор работает с непосредственно охватывающим его циклом. Но если нужно перескочить через один или несколько уровней, придется использовать метку, чтобы указать нужный вам цикл. Эта метка не обязана попадать в вашу лексическую область видимости, хотя, вероятно, ей следовало бы там находиться. На самом деле метка может находиться в любом месте вашей динамической области видимости. Если при этом приходится покидать eval или подпрограмму, Perl выдает предупредительное сообщение (если его об этом попросить).

Функция может иметь сколько угодно операторов return, а цикл может иметь сколько угодно операций управления циклом. Не следует считать это плохим тоном или некрасивым стилем. На заре структурного программирования некоторые настаивали, что на каждый цикл или подпрограмму должен приходиться строго один вход и один выход. Единственный вход – и на сегодня остается правильной идеей, однако концепция одного выхода заставляет нас писать большие объемы неестественного кода. Задачей программирования часто является обход дерева решений. Дерево решений обычно начинается с одного ствола и заканчивается многочисленными листьями. Вставляйте в код столько выходов из циклов (и возвратов из функций), сколько свойственно решаемой вами задаче. Если области видимости переменных определены разумно, то в нужный момент все автоматически очищается – независимо от того, каким образом выполняется выход из блока.

Операция last осуществляет незамедлительный выход из соответствующего цикла. Блок continue, если он есть, при этом не выполняется. В следующем примере осуществляется «катапультирование» из цикла после обнаружения первой пустой строки:

```
LINE: while (<STDIN>) {
    last LINE if /^$/;          # выйти после обработки почтового заголовка
```

```
}
```

Операция next пропускает оставшуюся часть текущей итерации цикла и начинает следующую. Если в цикле есть блок continue, он выполняется перед очередным вычислением условия цикла, точно так же, как третья составляющая трехчастного цикла for. Поэтому continue можно применить для приращения переменной цикла, даже если текущая итерация цикла была прервана операцией next:

```
LINE: while (<STDIN>) {
    next LINE if /^#/;         # пропустить комментарии
    next LINE if /^$/;         # пропустить пустые строки
    ...
} continue {
    $count++;
}
```

Оператор redo перезапускает блок цикла без повторной проверки условия. Блок continue, если он есть, не выполняется. Этот оператор часто используется в программах, которым требуется «обмануть» себя, читая данные. Предположим, что обрабатывается файл, в котором строка может завершаться обратной косой чертой, означающей продолжение записи на следующей строке. Вот как в этом случае можно использовать redo:

```
while (<>) {
    chomp;
    if (s/\\$//) {
        $_ .= <>;
        redo unless eof;       # не читать после eof каждого файла
    }
    # теперь обработать $_
}
```

Это обычное сокращение Perl для более явного (и скучного) варианта:

```
LINE: while (defined($line = <ARGV>)) {
    chomp($line);
    if ($line =~ s/\\$//) {
        $line .= <ARGV>;
        redo LINE unless eof(ARGV);
    }
    # теперь обработать $line
}
```

Вот пример из реальной программы, использующей все три операции управления циклом. Хотя данная стратегия разбора аргументов командной строки реже применяется сейчас, когда с Perl поставляются модули Getopt::*,[1] она все же представляет собой прекрасную иллюстрацию использования операций управления циклом для вложенных циклов с именами:

```
ARG: while (@ARGV && $ARGV[0] =~ s/^-(?=.)//) {
    OPT: for (shift @ARGV) {
```

[1] Сравнение основных модулей для разбора аргументов командной строки вы найдете в книге «Mastering Perl».

```
        m/^$/       && do {                          next ARG };
        m/^-$/      && do {                          last ARG };
        s/^d//      && do { $Debug_Level++;          redo OPT };
        s/^l//      && do { $Generate_Listing++,     redo OPT };
        s/^i(.*)//  && do { $In_Place = $1 || ".bak"; next ARG };
        say_usage("Неизвестный ключ: $_");
    }
}
```

Еще одно замечание относительно операций управления циклом. Как можно было заметить, мы не называем их операторами. Это связано с тем, что они не являются инструкциями, хотя, как всякое выражение, могут использоваться в таком качестве. Можно даже представить их себе как унарные операции, которым дозволено изменять порядок выполнения программы. Поэтому их допускается применять везде, где имеет смысл их использование в выражении. На самом деле операции управления циклом можно использовать даже там, где это не имеет смысла. Иногда можно встретить в программе такую ошибку:

```
open FILE, '<', $file
    or warn "Невозможно открыть $file: $!\n", next FILE, # НЕВЕРНО
```

Намерения благие, но next FILE **будет интерпретироваться как аргумент списочного оператора** warn. **Поэтому** next **выполнится раньше, чем** warn **получит возможность выдать предупреждение. В данном случае ситуацию легко исправить, обратив списочный оператор** warn **в вызов функции** warn **с помощью надлежащим образом расставленных скобок:**

```
open FILE, '<', $file
    or warn("Невозможно открыть $file: $!\n"), next FILE; # okay
```

Однако вы, возможно, сочтете следующий код более прозрачным:

```
unless (open FILE, '<', $file) {
    warn "Невозможно открыть $file  $!\[illegible]
    next FILE;
}
```

Голые блоки в роли циклов

Отдельный блок, независимо от наличия у него метки, семантически эквивалентен циклу с единственной итерацией. А это позволяет нам использовать last **для выхода из такого блока или** redo **для его повторного выполнения.[1] Заметьте, что это не так для блоков в** eval {}, sub {} **и, ко всеобщему удивлению,** do {}. **Эти три конструкции – не блоки циклов, поскольку сами по себе вообще не являются блоками (*BLOCK*); стоящее впереди ключевое слово делает их просто термами в выражении – такими, где по случаю оказался блок кода. Поскольку они не являются блоками циклов, им нельзя присвоить метку или применить к ним операции управления циклом. Операции управления циклом можно использовать только в настоящих циклах, так же как** return **– только в подпрограмме (ну, и еще в** eval**).**

---

1 **По причинам, которые, возможно, прояснятся (а возможно, и нет) после некоторых размышлений, оператор** next **тоже осуществляет выход из незацикленного блока. Однако есть одно отличие:** next **выполнит блок** continue, **а** last **– нет.**

Операции управления циклом не работают также в if и unless, поскольку это не циклы. Но всегда можно вставить дополнительную пару фигурных скобок, чтобы создать голый блок, который *считается* циклом:

```
if (/pattern/) {{
    last if /alpha/;
    last if /beta/;
    last if /gamma/;
    # какие-то действия, если все еще в if()
}}
```

Вот каким образом блок обеспечивает операциям управления циклом возможность работать с конструкцией do {}. Чтобы использовать next или redo в do, поместите внутрь голый блок:

```
do {{
    next if $x == $y;
    # здесь какие-то действия
}} until $x++ > $z;
```

Чтобы применить last, придется выразиться более прозрачно:

```
{
    do {
        last if $x = $y ** 2;
        # здесь какие-то действия
    } while $x++ <= $z;
}
```

А если требуется обеспечить возможность работы с обеими операциями управления циклом, придется обозначить эти блоки метками, чтобы их можно было различать:

```
DO_LAST: {
    do {
DO_NEXT:     {
                 next DO_NEXT if $x == $y;
                 last DO_LAST if $x = $y ** 2;
                 # здесь какие-то действия
             }
        } while $x++ <= $z
    }
```

Но, дойдя до такой конструкции (а то и раньше), лучше всего организовать обычный бесконечный цикл с last в конце:

```
for (;;) {
    next if $x == $y;
    last if $x = $y ** 2;
    # здесь какие-то действия
    last unless $x++ <= $z;
}
```

Выбор тем с помощью циклов

Perl не ограничивается одним оператором проверки соответствия указанной теме. Помимо оператора given, для этой цели можно также использовать оператор

foreach. Например, вот один из способов подсчитать количество вхождений строки в массив:

```
use v5.10.1,
my $count = 0,
for (@array) {
    when ("FNORD") { ++$count }
}
print "\@array содержит $count копий строки 'FNORD'\n";
```

Или, при использовании более свежей версии Perl:

```
use v5.14;
my $count = 0;
for (@array) {
    ++$count when "FNORD';
}
print "\@array содержит $count копий строки 'FNORD'\n";
```

В конце всех блоков when внутри цикла foreach неявно выполняется оператор break, который внутри циклов эквивалентен операции next. Если вас интересует только первое совпадение, такое поведение можно переопределить посредством явной операции last.

Оператор when действует, только если тема хранится в переменной $_, поэтому в таких случаях не получится использовать переменные цикла: если обращение к переменной происходит, это должна быть переменная $_:

```
for my $_ (@answers) {
    say "Жизнь, Вселенная и все сущее!" when 42;
}
```

Оператор goto

Не по малодушию (но и не с легким сердцем) Perl все же поддерживает оператор goto, который бывает трех видов: goto *LABEL*, goto *EXPR* и goto &*NAME*.

goto *LABEL* находит оператор с меткой *LABEL* и продолжает выполнение с него. Данную форму нельзя применять для перехода в какую-либо конструкцию, требующую инициализации, например, подпрограмму или цикл foreach. Нельзя также использовать ее для перехода в конструкцию, удаленную в результате оптимизации (см. главу 16). Она позволяет перейти почти в любое другое место в текущем блоке или любом блоке динамической области видимости (т.е. блоке, из которого был произведен вызов). Посредством goto можно даже выходить из подпрограмм, но обычно лучше делать это другими средствами. Автор Perl никогда не испытывал потребности использовать эту форму goto (хотя был вынужден сделать это в тестах, проверяющих работу этого оператора).

Форма goto *EXPR* является просто обобщением goto *LABEL*. Предполагается, что результат вычисления выражения *EXPR* представляет собой имя метки, местонахождение которой, очевидно, должно динамически определяться интерпретатором. В результате можно применять вычисляемые переходы, как в FORTRAN, что, если требуется облегчить сопровождение кода, не обязательно окажется хорошей идеей:

```
goto(("FOO", "BAR", "GLARCH")[$i]); # в надежде, что 0 <= i < 3

@loop_label = qw/FOO BAR GLARCH/,
goto $loop_label[rand @loop_label]; # случайная телепортация
```

В большинстве подобных случаев обычно гораздо (*гораздо*) более правильным решением является применение структурных механизмов управления последовательностью выполнения (next, last и redo), а не использование goto. В некоторых приложениях разумнее использовать хеш со ссылками на функции или пары eval и die для обработки исключительных ситуаций.

Форма goto &*NAME* является исключительно волшебной, и она достаточно далека от обычного goto, чтобы избавить применяющих ее от бесчестья, уготованного историей поклонникам goto. Она вызывает указанную подпрограмму вместо подпрограммы, выполняющейся в данный момент. Такое свойство этого оператора используется подпрограммами AUTOLOAD, чтобы загрузить другую подпрограмму, а затем представить дело так, будто та, другая подпрограмма и была вызвана изначально. После goto-перехода даже функция caller не сможет определить, что первой была вызвана данная подпрограмма. Все модули autouse, AutoLoader и SelfLoader используют эту стратегию, чтобы определять функции при первом к ним обращении, а затем передавать в них управление, так что никто никогда и не заметит, что этих функций в начале не было. Данная операция не особенно дешевая, так что не считайте ее эквивалентом хвостовой оптимизации.

Окаменевшие switch/case

В первые двадцать лет жизни Perl этот язык официально не имел оператора switch или case. До появления оператора given в v5.10 программисты были вынуждены изобретать собственные структуры ветвления на основе голых блоков и однопроходных циклов foreach. Вот один пример:

```
SWITCH: {
    if (/^abc/) { $abc = 1; last SWITCH }
    if (/^def/) { $def = 1; last SWITCH }
    if (/^xyz/) { $xyz = 1; last SWITCH }
    $nothing = 1;
}
```

а вот другой:

```
SWITCH: {
    /^abc/     && do { $abc = 1; last SWITCH; };
    /^def/     && do { $def = 1; last SWITCH; };
    /^xyz/     && do { $xyz = 1; last SWITCH; };
    $nothing = 1;
}
```

или даже просто:

```
if    (/^abc/) { $abc = 1 }
elsif (/^def/) { $def = 1 }
elsif (/^xyz/) { $xyz = 1 }
else           { $nothing = 1 }
```

А в следующем примере обратите внимание на то удобное обстоятельство, что операции last игнорируют блоки do {}, которые не являются циклами, и осуществляют выход из главного цикла:

```
for ($very_nasty_long_name[$i++][$j++]->method()) {
    /один шаблон/       and do { push @flags, '-e'; last };
    /другой/            and do { push @flags, '-h'; last };
    /третий/            and do {                    last };
    die "неизвестное значение: '$_'";
}
```

Эти идиомы вы будете время от времени встречать в старых программах на Perl, потому что до появления given цикл for был единственной возможностью реализовать выбор из множества тем.

Независимо от того, какая конструкция применяется для выбора тем, если есть возможность указать проверяемое значение лишь единожды, это здорово облегчает набор текста и, соответственно, снижает вероятность появления опечаток. Устраняется также возможность получения побочного эффекта от повторного вычисления выражения.

В простых случаях можно также каскадировать оператор ?:. В следующем примере мы снова прибегаем к оператору for и его свойству создания псевдонимов, чтобы улучшить читаемость повторных сравнений:

```
for ($user_color_preference) {
    $value = /red/   ? 0xFF0000 :
             /green/ ? 0x00FF00 :
             /blue/  ? 0x0000FF :
                       0x000000 ; # черный, если нет совпадений
}
```

Однако во многих ситуациях бывает более уместно создать собственный хеш и быстро найти в нем ответ по ключу. В отличие от только что показанного каскадирования условных операторов, хеш может содержать неограниченное число элементов, и время поиска последнего из них ничуть не больше, чем первого. Более того, хеш позволяет добавлять варианты выбора прямо во время выполнения. Недостаток в том, что можно использовать только точный поиск, но не соответствие шаблону. Если есть такой хеш:

```
%color_map = (
    azure      => 0xF0FFFF,
    chartreuse => 0x7FFF00,
    lavender   => 0xE6E6FA,
    magenta    => 0xFF00FF,
    turquoise  => 0x40E0D0,
);
```

точный поиск строки выполняется быстро, и по-прежнему есть возможность добавить значение по умолчанию:

```
$value = $color_map{ lc $user_color_preference } || 0x000000;
```

Даже сложные многократно ветвящиеся операторы (где в каждом случае требуется выполнить несколько различных операторов) можно превратить в быстрый поиск по хешу. Нужно только использовать хеш ссылок на функции, которые

в Perl являются законными типами данных. Как это делается, рассказано в разделе «Хеши функций» главы 9.

Ко всему вышесказанному остается только добавить, что структуры выбора из нескольких вариантов – не всегда самый удачный инструмент вашего арсенала. Наиболее расширяемый способ реализации поиска заключается в использовании полиморфизма объектов. Если вам не терпится узнать об этом поскорее, загляните в главу 12.

Ниже приводится пример еще одной ужасающий структуры выбора:

```
goto $data;
ABC: $foo++; goto end;
DEF: $bar++; goto end;
XYZ: $baz++, goto end;
end:
```

Да, она работает, но... очень медленно, и если ни одна из меток не совпадет с искомым значением, поиск будет продолжен вплоть до конца программы и, если вам повезет, приведет к сбою. Однако существуют более удачные способы вызвать аварийное завершение программы, и об одном из них рассказывается в следующем разделе.

Оператор многоточия

В Perl версии v5.12 появился оператор голое многоточие «. », играющий роль заглушки, или метки-заполнителя для еще не написанного кода программы. Не путайте его с оператором диапазона, . . Perl обычно их не путает, потому что в большинстве случаев может четко сказать, когда ожидает встретить тот или иной оператор – в большинстве случаев, но не всегда.

Когда Perl встречает оператор многоточия на этапе компиляции, то принимает его молча. Однако позднее, если программа попытается выполнить этот оператор, Perl возмущенно возбудит исключение с текстом "Unimplemented":

```
sub unimplemented { ..  }
eval { unimplemented() };
if ($@ =~ /^Unimplemented/) {
    say "Возникло исключение Unimplemented'
}
```

Оператор многоточия разрешается использовать только на этапе компиляции и только как законченную инструкцию (однако допускается использовать модификатор оператора). Все следующие примеры являются допустимыми примерами использования оператора многоточия:

```
{ ... }
sub foo {     }
...;
eval { ..  };
... unless defined &dispatcher,
sub somemeth {
    my $self = shift;
    ...;
}
```

```
$x = do {
    my $n;
    ...;
    say "Hurrah!",
    $n;
};
```

Однако оператор ... не может быть частью выражения или составного оператора, поскольку существует также трехточечная (...) версия оператора диапазона (см. раздел «Операторы диапазона» в главе 3). По этой причине все следующие примеры содержат ошибки синтаксиса:

```
print ...,                                      # НЕВЕРНО
open(my $fh, ">", "/dev/passwd") or ...;        # НЕВЕРНО
if ($condition && ... ) { say "Howdy" };        # НЕВЕРНО
```

В некоторых ситуациях Perl может не различать выражения и операторы. Например, голый блок и объявление анонимного хеша выглядят одинаково, если внутри скобок нет чего-то еще, позволяющего Perl понять, с чем именно он имеет дело:

```
@transformed = map { ... } @input; # НЕВЕРНО: ошибка синтаксиса
```

Одно из решений этой проблемы заключается в использовании ; внутри блока, чтобы подсказать Perl, что { ... } – это блок, а не анонимный хеш:

```
@transformed = map {; ... } @input; # ; однозначно определяет оператор многоточия
@transformed = map { ...; } @input; # ; однозначно определяет оператор многоточия
```

Программисты в разговорах между собой называют такие знаки пунктуации «yada-yada» («и тому подобное», или «и прочее»), но вы можете использовать технический термин «многоточие», если желаете произвести впечатление на особо впечатлительных. Perl не распознает версию многоточия в виде символа Юникода U+2026 HORIZONTAL ELLIPSIS, но может быть, в один прекрасный день...

Глобальные объявления

Объявления подпрограмм и форматов являются глобальными. Независимо от того, куда они помещены, объявляемые ими элементы являются глобальными (локальными для пакета, но пакеты глобальны для программы, поэтому все, что находится в пакете, видимо в любом месте). Глобальное объявление можно поместить в любое место, где может находиться оператор, но оно не оказывает влияния на выполнение основной последовательности операторов: действие объявлений сказывается только на этапе компиляции.

Это значит, что нельзя создавать условные объявления подпрограмм или форматов, скрывая их от компилятора в условных операторах этапа выполнения типа if, поскольку только интерпретатор обращает внимание на условное выполнение этих операторов. Объявления подпрограмм и форматов (а также объявления use и no) видны компилятору независимо от того, где они находятся.

Глобальные объявления обычно помещаются в начале или конце программы либо отдельно в другом файле. Однако при объявлении переменных с лексической областью видимости (см. следующий раздел) нужно обеспечить, чтобы определение формата или подпрограммы попало в область видимости объявлений тех переменных, доступ к которым вы хотите иметь в этом определении.

Заметьте, что мы незаметно перешли в разговоре с объявлений на определения. Иногда полезно разделить *определение* (*definition*) подпрограммы и ее *объявление* (*declaration*). Единственная синтаксическая разница между ними в том, что определение включает блок, содержащий код, который должен быть выполнен, а в объявлении его нет. (Определение подпрограммы действует как ее объявление, если объявления нет в области видимости.) Отделение определения от объявления позволяет поместить объявление подпрограммы в начало файла, а определение – в конец (при этом ваши объявления переменных с лексической областью видимости удачно располагаются посередине):

```
sub count (@);          # Теперь компилятор знает, как вызывать count().
my $x;                  # Теперь компилятор знает о лексической переменной.
$x = count(3,2,1);      # Компилятор может проверить вызов функции.
sub count (@) { @_ }    # Теперь компилятор знает, что означает count().
```

Как показывает этот пример, присутствие фактических определений подпрограмм не требуется, чтобы их вызовы могли быть скомпилированы (более того, определение можно даже отложить до первого вызова, если применяется автозагрузка), однако объявление подпрограмм оказывает компилятору различную помощь и дает вам большую свободу в способах их вызова.

Объявление подпрограммы позволяет начать использовать ее с точки объявления, без скобок, как если бы это был встроенный оператор. (В последнем примере мы использовали скобки для вызова count, но фактически в этом нет необходимости.) Чтобы объявить подпрограмму, не определяя ее, скажите просто:

```
sub myname;
$me = myname $0 or die "Не могу найти myname";
```

Такое голое объявление делает функцию списочным, а не унарным оператором, поэтому будьте внимательны и применяйте or, а не ||. Оператор || слишком сильно связывает аргументы, чтобы использовать его после списочных операторов, хотя аргументы списочного оператора всегда можно заключить в круглые скобки, превратив его снова в нечто, более похожее на вызов функции. Альтернативным вариантом является использование прототипа ($), позволяющее превратить подпрограмму в унарный оператор:

```
sub myname ($);
$me = myname $0 || die "Не могу найти myname";
```

Теперь Perl разберет код так, как вы ожидаете, но все же следует выработать привычку использовать в такой ситуации скобки. Подробнее о прототипах читайте в главе 7.

Вы *обязаны* определить подпрограмму в каком-то месте, иначе на этапе выполнения получите ошибку, указывающую на вызов процедуры, которая не определена. Помимо самостоятельного определения подпрограммы есть несколько способов получать определения из других мест.

Определения можно загружать из других файлов с помощью простого оператора require. В Perl 4 это был лучший способ загрузки файлов, но с ним связаны две проблемы. Во-первых, другой файл обычно помещает имена подпрограмм в пакет (таблицу имен) по собственному выбору, а не в ваши пакеты. Во-вторых, обращение к require происходит на этапе выполнения, т.е. слишком поздно, чтобы

предоставлять объявления файлу, вызывающему require. Иногда, впрочем, отложенная загрузка – это именно то, что доктор прописал.

Более удобный способ загрузки объявлений и определений предоставляет директива use, которая, по сути, вызывает require для модуля на этапе компиляции (поскольку use рассматривается как блок BEGIN) и позволяет импортировать некоторые из объявлений модуля в вашу программу. Так что use можно рассматривать как своего рода глобальное объявление, ведь на этапе компиляции эта инструкция импортирует имена в ваш собственный (глобальный) пакет так, как если бы вы объявили их в основном тексте программы. В разделе «Таблицы символов» главы 10 приведено описание механизма низкого уровня, действующего при импорте из одного пакета в другой; в главе 11 – описание настройки импорта и экспорта в модуле; в главе 16 – описание BEGIN и родственных ему CHECK, INIT и END, которые тоже в некотором роде служат глобальными объявлениями, поскольку работа с ними происходит на этапе компиляции и может создавать глобальные эффекты.

Объявления с областью видимости

Как и глобальные объявления, объявления с лексической областью видимости оказывают воздействие на этапе компиляции. В отличие от глобальных объявлений, объявления с лексической областью видимости действуют только от точки, где находится объявление, до конца самой глубокой из охватывающих областей видимости (блока, файла или eval – что встретится раньше). Поэтому мы говорим: «с лексической областью видимости», хотя «с текстовой областью видимости» было бы, вероятно, точнее, поскольку «лексическая область видимости» имеет мало общего со словарями (лексиконами). Однако компьютерные специалисты во всем мире знают, что означает «лексическая область видимости» («lexically scoped»), поэтому мы сохраняем здесь это название.

Perl также поддерживает объявления с динамической областью видимости. *Динамическая область* видимости (dynamic scope) тоже распространяется до конца самого глубокого из охватывающих блоков, но в этом случае «охватывание» динамически определяется на этапе выполнения, а не текстуально, на этапе компиляции. Иными словами, динамическая вложенность блоков определяется вызовом одного блока другим, а не включением одного в состав другого. Эта вложенность динамических областей может отчасти коррелировать с вложенностью лексических областей, но, вообще говоря, они не идентичны, особенно если вызываются какие-либо подпрограммы.

Мы уже говорили, что в некоторых отношениях use можно рассматривать в качестве глобального объявления, но в других отношениях use имеет лексическую область видимости. В частности, use не только импортирует имена из пакета, но также реализует волшебные подсказки для компилятора, называемые *прагмами*. Прагмы в большинстве своем имеют лексическую область видимости; к таким относится, например, директива strict, которую мы время от времени упоминаем. См. далее раздел «Прагмы». (Отсюда следует, что, если какая-то особенность языка окажется неявно включена прагмой use v5.14 в начале файла, она будет действовать до конца файла, даже при переключении пакетов.)

Объявление пакета (package), как ни странно, имеет лексическую область видимости, несмотря на то, что пакет представляет собой глобальный объект. Но объявление пакета просто объявляет имя пакета по умолчанию для оставшейся части

охватывающего блока или, если вы включили после объявления package NAMESPACE необязательный блок (BLOCK), до конца этого блока. Поиск необъявленных идентификаторов[1] производится в этом пакете. В некотором смысле пакет вовсе и не объявляется, но начинает существовать, когда вы ссылаетесь на нечто, принадлежащее этому пакету. Это вполне в духе Perl.

Объявления переменных с областью видимости

Оставшаяся часть главы посвящена в основном использованию глобальных переменных. Или, скорее, тому, что *не следует* использовать глобальные переменные. Есть несколько объявлений, помогающих обойтись без глобальных переменных – или, по крайней мере, применять их разумно.

Мы уже упоминали объявление package, давным-давно введенное в Perl для целей распределения глобальных переменных по пакетам. Для некоторых типов переменных это работает довольно хорошо. Пакеты используются библиотеками, модулями и классами для хранения данных своих интерфейсов (и некоторых своих полузакрытых данных), чтобы избежать конфликтов с одноименными переменными и функциями в основной программе или других модулях. Так, запись $Some::stuff[2] означает, что автор обращается к скалярной переменной $stuff из пакета Some. См. главу 10.

Ограничься мы только этим, программы на Perl по мере их развития быстро становились бы громоздкими. К счастью, имеющиеся в Perl три объявления области видимости позволяют легко создавать совершенно закрытые переменные (с помощью my или state), предоставлять избирательный доступ к глобальным переменным (посредством our) и обеспечивать временные значения для глобальных переменных (с использованием local):

```
my $nose;
our $House;
state $troopers = 0;
local $TV_channel;
```

Если в объявлении более одной переменной, список должен быть заключен в круглые скобки.

```
my ($nose, @eyes, %teeth);
our ($House, @Autos, %Kids);
state ($name, $rank, $serno);
local (*Spouse, $phone{HOME});
```

В объявлениях my, state и [illegible] допускаются только простые скалярные переменные, переменные массивов и хешей, тогда как state позволяет инициализировать простые скалярные переменные (которые, впрочем, могут содержать ссылки на все, что угодно), но не массивы или хеши. Поскольку local – не настоящее объявление, его ограничения не столь строгие: можно делать локальными, с инициализацией или без нее, целые записи таблицы имен typeglob и отдельные элементы

---

[1] А также неквалифицированные имена подпрограмм, дескрипторов файлов, дескрипторов каталогов и форматов.

[2] Или архаическую $Some'stuff, использование которой за рамками поэзии на Perl не приветствуется.

или срезы массивов и хешей. Каждый из этих модификаторов имеет свой «режим заключения» модифицируемых переменных. Несколько упрощая, можно сказать, что our ограничивает область видимости имен, local ограничивает область видимости значений, а my ограничивает область видимости, как имен, так и значений. (Объявление state действует подобно объявлению my, но несколько иначе понимает область видимости.) Каждой из этих конструкций может быть выполнено присваивание, хотя они различаются по тому, что фактически делают со значениями, поскольку имеют различные механизмы хранения значений. Они также несколько различаются, если вы *не присваиваете* (как было сделано выше) им значений: объявление my или local влечет присвоение указанным переменным начальных значений undef или (), в зависимости от типа, тогда как our оставляет текущее значение соответствующей глобальной переменной неизменным. А переменные, объявленные как state, получают значение, которое они имели при выполнении этого фрагмента кода в прошлый раз.

Синтаксически, our, state и local являются просто модификаторами левостороннего выражения, так что они похожи на прилагательные. Когда выполняется присваивание модифицированному таким образом l-значению, модификатор не влияет на то, рассматривается ли l-значение как скаляр или как список. Чтобы узнать, как будет работать присваивание, можно просто представить себе, что модификатор отсутствует. Поэтому любой из вариантов

```
my ($foo) = <STDIN>;
my @array = <STDIN>;
```

предоставляет списочный контекст для правой части, в то время как

```
my $foo = <STDIN>;
```

предоставляет скалярный контекст.

Модификаторы объявлений связывают аргументы более прочно (с более высоким приоритетом), чем запятая. В следующем примере ошибочно модифицируется одна, а не две переменных, поскольку список, следующий за модификатором, не заключен в круглые скобки.

```
my $foo, $bar = 1;              # НЕВЕРНО
```

Это равносильно следующему:

```
my $foo;
$bar = 1;
```

Если действует прагма strict, Perl выдаст сообщение об ошибке, так как переменная $bar оказывается необъявленной.

В целом, лучше всего установить для переменной минимальную область видимости, позволяющую эту переменную дельно применить. Поскольку переменные, объявленные в операторе управления логикой программы, видимы только в блоке, которым управляет этот оператор, их область видимости уменьшается. Благодаря этому код становится еще и проще читать.

```
sub check_warehouse {
    for my $widget (our @Current_Inventory) {
        say "Сегодня у меня на складе есть $widget.";
    }
}
```

Чаще всего встречается объявление my, создающее переменные с лексической областью видимости, имена и значения которых хранятся во временной памяти текущей области видимости и не доступны глобально. Всегда используйте my, если только не знаете наверняка, зачем вам нужен иной модификатор. А если вы хотите обеспечить ту же степень закрытости переменной, но сохранить ее значение до следующего вызова, используйте state.

Очень близко к этим двум объявление our, которое вводит в текущую область видимости имя с лексической областью видимости, но при этом фактически ссылается на глобальную переменную, к которой при желании может обратиться каждый. Иными словами, это глобальная переменная, маскирующаяся под лексическую.

Другая форма глобальной области видимости, *динамическая область видимости*, применяется к переменным с модификатором local, которые, несмотря на слово «local», являются на самом деле глобальными переменными и не имеют никакого отношения к локальной временной памяти. (Имя temp точнее отражало бы назначение этого модификатора, поскольку он временно меняет значение существующей переменной. В один прекрасный день вы можете даже увидеть слово temp в программах на Perl 5, если мы заимствуем это ключевое слово из Perl 6.)

Вновь объявленная переменная my (или значение, в случае local) недоступна до оператора, следующего *после* оператора, содержащего объявление. Благодаря этому можно создать зеркальное отражение переменной, как показано ниже:

```
my $x = $x;
```

Эта строка инициализирует новую внутреннюю переменную $x текущим значением существующей переменной $x, где под существующей подразумевается переменная $x в глобальной или лексической области видимости.

Объявление имени лексической переменной скрывает ранее объявленную лексическую переменную с тем же именем, которая объявлена в этой же или во внешней области видимости (хотя в этом случае можно получить предупреждение, если включен строгий режим). Оно также скрывает любую неквалифицированную глобальную переменную с тем же именем, но к глобальной переменной всегда можно обратиться, явно указав полное квалифицированное имя, включающее имя пакета, например, $PackageName::varname.

Переменные с лексической областью видимости: my

Чтобы помочь избежать хлопот, связанных с поддержкой глобальных переменных, Perl предоставляет переменные с лексической областью видимости, часто для краткости называемые *лексическими* (*lexicals*). В отличие от глобальных, лексические переменные гарантируют приватность. Если ссылки на эти переменные не выставляются напоказ, что позволило бы манипулировать ими косвенным образом, можете быть уверены, что всякий доступ к этим закрытым переменным ограничен кодом в одной ограниченной, непрерывной области вашей программы, которую легко найти. В конце концов, потому мы и выбрали ключевое слово my (мой).

В последовательности операторов могут содержаться объявления переменных с лексической областью видимости. Такими объявлениями обычно предваряют последовательность операторов, но это не обязательное требование – вы можете просто украсить первое обращение к переменной объявлением my (при условии,

что это объявление находится в самой внешней области использования переменной). Помимо объявления имен переменных на этапе компиляции, эти объявления действуют как обычные операторы на этапе выполнения: все они обрабатываются внутри последовательности операторов, как если бы были обычными операторами, а не объявлениями:

```
my $name = "fred";
my @stuff = ("car", "house", "club");
my ($vehicle, $home, $tool) = @stuff;
```

Эти лексические переменные совершенно скрыты от всего мира за пределами непосредственно охватывающей их области видимости. В отличие от ситуации с динамической областью видимости, создаваемой local (см. следующий раздел), лексические переменные скрыты от любых подпрограмм, вызываемых из их области видимости. Это верно даже в отношении той же самой подпрограммы, вызываемой из себя самой или из любого другого места: каждый экземпляр подпрограммы получает собственную «временную память» для лексических переменных. Однако подпрограммы, объявленные в области видимости лексической переменной, видят эту переменную, как и любой другой код в этой области видимости.

В отличие от области видимости блока, области видимости файла не могут быть вложенными; «охват», по крайней мере, текстуально, не имеет места. Если вы загружаете код из отдельного файла с помощью do, require или use, то код в этом файле не имеет доступа к вашим лексическим переменным, равно как и вы не имеете доступа к лексическим переменным из этого файла.

Однако любая область видимости внутри файла (и даже сам файл) действует по правилам. Часто удобно иметь области видимости более широкие, чем определения функций, так как это позволяет организовать совместный доступ к закрытым переменным для ограниченного набора подпрограмм. Так создаются переменные, которые программист на языке C назвал бы статическими:

```
{
    my $state = 0;

    sub on     { $state = 1 }
    sub off    { $state = 0 }
    sub toggle { $state = !$state }
}
```

Оператор eval *STRING* тоже действует как вложенная область видимости, поскольку код в eval может видеть лексические переменные вызвавшей его области (если их имена не скрыты идентичными объявлениями в собственной области видимости eval). Аналогично анонимные подпрограммы могут обращаться ко всем лексическим переменным охватывающей их области видимости; когда это происходит, их называют *замыканиями* (*closures*).[1] Объединим эти два замечания: если блок выполняет eval над строкой и создает при этом анонимную подпрограмму, эта подпрограмма становится замыканием с полным доступом к лексическим пе-

[1] Мнемоническое правило основывается на общем элементе в «en*clo*sing scope» (охватывающая область видимости) и «*clo*sure» (замыкание). (В действительности определение замыкания происходит от математического понятия, означающего полноту набора величин и операции над этими величинами.)

ременным как eval, так и блока, даже после выхода из eval и блока. См. раздел «Замыкания» главы 8.

Лексические переменные, сохраняющие свое значение: state

Переменная с декларацией state является лексической, подобно переменой my. Единственное отличие в том, что state-переменные никогда не инициализируются повторно, в отличие от my-переменных, которые инициализируются каждый раз, когда происходит вход в непосредственно охватывающий блок. Обычно переменные state используются с целью дать функции частную переменную, сохраняющую свое значение между вызовами функции.

Переменные state доступны, только если действует прагма use feature "state". Она включается автоматически, если с помощью объявления use указана версия Perl не ниже v5.10:

```
use v5.14;
sub next_count {
    state $counter = 0; # инициализация выполняется только в первый раз
    return ++$counter;
}
```

В отличие от переменных my, переменные state в настоящее время могут быть только скалярными – они не могут хранить массивы или хеши. Это может показаться более серьезным ограничением, чем в действительности, ведь вы всегда можете сохранить в переменной вида state ссылку на массив или хеш:

```
use v5.14;
state $bag    = { };
state $vector = [ ];
...
unless ($bag->{$item}) { $bag->{$item} = 1 }
...
push @$vector, $item,
```

Глобальные объявления с лексической областью видимости: our

В былые времена, до появления use strict, программы на Perl могли напрямую обращаться к глобальным переменным. Объявление our являет собой более современный метод такого доступа. Оно имеет лексическую область видимости, поскольку действует только до конца текущей области видимости. Но, в отличие от лексической области видимости my или динамической области видимости local, our не изолирует ничего в текущей лексической или динамической области видимости. Напротив, оно обеспечивает доступ к глобальной переменной в текущем пакете, маскируя одноименные лексические переменные, которые, в противном случае, закрыли бы от вас эту глобальную переменную. В этом отношении переменные our действуют точно так же, как переменные my.

Если поместить объявление our за пределами всех блоков, заключенных в фигурные скобки, оно действует до конца текущей единицы компиляции. Однако часто

его помещают в начале подпрограммы, чтобы указать на необходимость обращения к глобальной переменной:

```
sub check_warehouse {
    our @Current_Inventory;
    my $widget;
    foreach $widget (@Current_Inventory) {
        say "Сегодня у меня на складе есть $widget.";
    }
}
```

Поскольку глобальные переменные живут дольше и видимы шире, чем закрытые, мы предпочитаем использовать для них более длинные и броские имена, чем для временных переменных. Одно лишь это правило, если ему старательно следовать, может столь же успешно отбить охоту к работе с глобальными переменными, как и использование `use strict`, особенно у тех, кто не особенно искусен в обращении с клавиатурой.

Повторные объявления `our` не имеют контекста вложенности. Каждое вложенное `my` создает новую переменную, а каждое вложенное `local` создает новое значение. Но всякий раз, когда используется `our`, речь идет *о той же самой* глобальной переменной, независимо от уровня вложенности. Если осуществляется присваивание переменной `our`, его результат сохраняет свое действие и после конца области видимости присваивания. Дело в том, что `our` не создает значений; оно просто предоставляет ограниченную форму доступа к глобальной переменной, срок жизни которой не ограничен:

```
our $PROGRAM_NAME = "waiter";
{
    our $PROGRAM_NAME = "server";
    # Вызываемый здесь код видит "server".

}
# Выполняемый здесь код по-прежнему видит "server"
```

Сравните все это с тем, что происходит с `my` или `local`, когда за пределами блока наружная переменная или значение снова становятся видимыми:

```
my $i = 10;
{
    my $i = 99;

}
# Компилируемый здесь код видит наружную переменную со значением 10.

local $PROGRAM_NAME = "waiter"
{
    local $PROGRAM_NAME = "server";
    # Код, выполняемый здесь, видит "server"

}
# Код, выполняемый здесь, снова видит "waiter".
```

Обычно имеет смысл делать только одно присваивание в объявлении our, вероятно, в самом начале программы или модуля, либо, реже, если вы предваряете our собственным local:

```
{
    local our @Current_Inventory = qw(bananas);
    check_warehouse(); # нет, бананов у нас нет :-)
}
```

(Но почему бы в этом случае просто не передать значение, как аргумент?)

Переменные с динамической областью видимости: local

Использование объявления local с глобальной переменной дает этой переменной временное значение при каждом выполнении local, но не оказывает влияния на глобальную видимость переменной. Когда программа достигает конца текущей динамической области видимости, временное значение отбрасывается и восстанавливается первоначальное значение. Но сама переменная всегда остается глобальной, и всего лишь получает временное значение на период выполнения данного блока. Если вызвать какую-либо другую функцию в то время, когда эта глобальная переменная содержит временное значение, и внутри функции обратиться к данной глобальной переменной, функция увидит временное значение переменной, а не первоначальное. Иными словами, эта другая функция находится в вашей динамической области видимости, хотя, предположительно, не в лексической области видимости.[1]

Этот процесс называется *образованием динамической области видимости*, потому что текущее значение глобальной переменной зависит от динамического контекста – от того, кто из предков в цепочке вызовов применил оператор local. Иначе говоря, вызывающий программный код может полностью контролировать значение, которое вы увидите.

Если вы встречаете объявление local, которое выглядит так:

```
{
    local $var = $newvalue;
    some_func();
    ...
}
```

то можете представлять его себе исключительно в терминах присваиваний этапа выполнения:

```
{
    $oldvalue = $var
    $var = $newvalue;
    some_func();
    ...
```

[1] Вот почему лексические области видимости иногда называют *статическими*: чтобы противопоставить их динамическим областям видимости и подчеркнуть их детерминированность на этапе компиляции. Не путайте это значение термина с тем, которое слово «статический» имеет в C и C++. Этот термин слишком сильно перегружен значениями, почему мы и стараемся его не использовать.

```
}
continue {
    $var = $oldvalue;
}
```

Разница в том, что при использовании local значение восстанавливается независимо от того, каким образом происходит выход из блока, даже если это досрочный возврат из данной области видимости.

Как и в случае применения my, можно инициализировать local копией той же самой глобальной переменной. Все изменения, происходящие с этой переменной во время выполнения подпрограммы (и всех других, вызываемых из нее, которые, разумеется, по-прежнему видят глобальную переменную с динамической областью видимости), будут отброшены при возврате из подпрограммы. Конечно, лучше всего снабдить свои действия комментариями:

```
# ПРЕДУПРЕЖДЕНИЕ: Изменения в этой динамической области видимости
# являются временными
local $Some_Global = $Some_Global;
```

После этого глобальная переменная по-прежнему видна всюду в программе независимо от того, была ли она явно определена с помощью our или возникла из ничего, или содержит значение local, подлежащее отбрасыванию при выходе из области видимости. В маленьких программках это не столь плохо, но в больших программах вы быстро потеряете возможность отслеживать, где в коде применяются все эти глобальные переменные. При желании можно запретить случайное использование глобальных переменных с помощью директивы use strict 'vars', описываемой в следующем разделе.

Хотя и my, и local предоставляют некоторую степень защиты, в целом предпочтительнее использовать my, а не local. Однако иногда применение local необходимо, чтобы временно изменить значение существующих глобальных переменных, например, перечисленных в главе 25. Лексическую область видимости могут иметь только буквенно-цифровые идентификаторы, а многие из этих специальных переменных не являются строго буквенно-цифровыми. Без local также не обойтись при внесении временных изменений в таблицу имен пакета, как показано в разделе «Таблицы имен» главы 10. Наконец, можно распространить действие local на один элемент или целый срез массива или хеша. Это допустимо, даже если массив или хеш являются лексической переменной, при этом режим динамической области видимости local накладывается поверх этих лексических переменных. Мы не станем здесь больше распространяться о семантике local. Дополнительные сведения можно получить из описания local в главе 27.

Прагмы

Многие языки программирования позволяют давать компилятору советы. В Perl эти советы передаются компилятору посредством объявления use. Вот некоторые из них:

```
use warnings;
use strict;
use integer;
use bytes;
```

```
use constant pi => ( 4 * atan2(1,1) );
```

Все прагмы Perl описаны в главе 29, но сейчас мы отдельно поговорим о некоторых из них, особенно полезных в связи с материалом, изложенным в данной главе.

Хотя некоторые прагмы представляют собой глобальные объявления, воздействующие на глобальные переменные в текущем пакете, в большинстве своем это объявления с лексической областью видимости, действие которых продолжается только до конца охватывающего блока, файла или `eval` (в зависимости от того, что встретится раньше). Прагму с лексической областью видимости можно отменить во вложенной области видимости с помощью объявления `no`, которое действует как `use`, но противоположным образом.

Управление выводом предупреждений

Чтобы показать, как это работает, мы поработаем с прагмой `warnings`, указывая Perl, когда следует выводить предупреждения о сомнительных приемах программирования:

```
use warnings;    # Включить вывод предупреждений отсюда и до конца файла
...
{
    no warnings; # Отключить вывод предупреждений до конца блока
    ...
}
# Здесь вывод предупреждений автоматически снова активизируется
```

Когда вывод предупреждений включен, Perl сообщает о переменных, использованных только один раз; объявлениях переменных, скрывающих другие объявления в той же области видимости; недопустимых преобразованиях строк в числа; использовании неопределенных значений в качестве допустимых строк или чисел; попытках записи в файлы, открытые только для чтения (или вообще не открытые); и многих других ситуациях, описанных в разделе справочного руководства *perldiag*.

Прагма `warnings` – предпочтительный способ управления выводом предупреждений. В старых программах этим целям служил ключ командной строки `-w` или состояние глобальной переменной `$^W`:

```
{
    local $^W = 0;
    ...
}
```

Гораздо правильнее применять прагмы `use warnings` и `no warnings`. Дело в том, что обработка прагмы выполняется на этапе компиляции, а сама прагма представляет собой лексическое объявление и, следовательно, не может влиять на код, на который не должно влиять, а кроме того (хотя мы не показали этого в наших простых примерах), позволяет тонко управлять отдельными классами сообщений. Дополнительные сведения о прагме `warnings`, в том числе информация о том, как преобразовать просто надоедливые сообщения в фатальные ошибки и как переопределить прагму, чтобы включить глобальный вывод предупреждений, даже если модуль требует обратного, можно найти в описании прагмы `warnings` в главе 29.

Управление использованием глобальных переменных

Еще одно распространенное объявление – многоцелевая прагма strict, одной из функций которой является управление использованием глобальных переменных. Обычно Perl позволяет создавать новые переменные (а нередко и «затаптывать» старые) путем простого их упоминания. По умолчанию, объявлять переменные вообще не требуется. Поскольку бесконтрольное применение глобальных переменных может сделать мучительным сопровождение больших программ и модулей, иногда желательно воспрепятствовать случайному их использованию. Чтобы содействовать предотвращению таких несчастных случаев, можно сказать:

```
use v5.14,          # Включить strict неявно.
use strict "vars", # Включить strict явно
```

Это означает, что с данного места и до конца охватывающей области видимости все упоминаемые переменные должны относиться либо к лексическим переменным, объявленным с помощью my, state или our, либо к переменным, которые явным образом разрешены глобально. В противном случае возникает ошибка компиляции. Глобальная переменная считается явно разрешенной в одном из следующих случаев:

- это одна из специальных переменных Perl, действующих всюду в программе (см. главу 25);
- она полностью квалифицирована именем своего пакета (см. главу 10);
- она импортирована в текущий пакет (см. главу 11);
- она маскируется под переменную с лексической областью видимости посредством объявления our (это основная причина, по которой мы ввели в Perl объявления our).

Конечно, всегда есть пятая альтернатива: если применение этой прагмы становится обременительным, можно просто отменить ее во внутреннем блоке с помощью:

```
no strict "vars"
```

Данная прагма позволяет также включить строгую проверку символического разыменования и случайного использования «голых» слов. Обычно просто говорят:

```
use strict;
```

чтобы включить все три ограничения, если они еще не были включены объявлением use v5.14 или подобным ему. Дополнительные сведения можно найти в описании strict в главе 29.

5

Поиск по шаблону

Встроенная в Perl поддержка сопоставления с шаблоном обеспечивает удобный и эффективный поиск в больших объемах данных. Если вы управляете огромным коммерческим порталом, сканирующим все существующие ленты новостей в поисках интересной информации; правительственной организацией, посвятившей себя анализу демографических данных (или генома человека); или же просто работаете в образовательном учреждении и пытаетесь организовать представление каких-то динамических данных на своем веб-сайте, идеальным инструментом для вас будет Perl – отчасти благодаря его возможностям работы с базами данных, но в большей степени благодаря возможностям поиска по шаблону. Если рассматривать «текст» в самом широком смысле этого слова, то, вероятно, 90% всего, что мы делаем, на 90% состоит из обработки текста. Это действительно основной талант языка Perl, и так было с самого начала; он упоминается даже в самом названии Perl: Practical *Extraction* and Report Language – практический язык *извлечения* данных и генерации отчетов. Шаблоны Perl предоставляют мощный способ просмотра гор «сырых» данных с целью извлечения из них полезной информации.

Шаблон задается путем создания *регулярного выражения* (*regular expression*, или *regex*). Механизм регулярных выражений Perl (в оставшейся части главы мы будем называть его просто «Механизм») берет это выражение и определяет, соответствует ли этот шаблон (и каким образом) вашим данным. Хотя чаще всего данные представляют собой текстовые строки, ничто не мешает применять регулярные выражения для поиска и замены в любых последовательностях байтов, включая и объекты, которые мы обычно считаем «двоичными» данными. Для Perl байты – это просто символы, порядковые значения которых, по стечению обстоятельств, меньше 256. (Подробнее об этом рассказывается в главе 6.)

Тех, кто уже сталкивался где-то с регулярными выражениями, мы должны предупредить, что в Perl регулярные выражения несколько иные. Во-первых, они не являются полностью регулярными в теоретическом смысле этого слова, поскольку их возможности значительно шире, чем у традиционных регулярных выражений, которые преподаются в курсах вычислительной техники. Во-вторых, они так часто применяются в Perl, что имеют свои особые переменные, операторы и соглашения по использованию кавычек, тесно интегрированные с языком, а не

просто прицеплены к платформе языка, как сторонняя библиотека. Программисты, не имеющие опыта работы с Perl, нередко тщетно ищут функции для поиска и замены вроде таких:

```
match( $string, $pattern );
subst( $string, $pattern, $replacement )
```

Но поиск и подстановка в Perl являются столь фундаментальными задачами, что заслужили быть представленными однобуквенными операторами: m/*PATTERN*/ и s/*PATTERN*/*REPLACEMENT*/, для краткости – m// и s///. Эти операции не только имеют краткий синтаксис, но и анализируются не как обычные операторы, а как строки в двойных кавычках; тем не менее они действуют как операторы, и так мы и будем их называть. На протяжении всей этой главы рассматривается применение этих операторов для поиска шаблонов в строке. Если какая-то часть строки согласуется с шаблоном, то мы будем говорить, что поиск успешен. Если поиск успешен, можно выполнить массу интересных операций. В частности, в случае применения s/// успешный поиск приводит к замене обнаруженной части строки тем, что было задано в качестве *REPLACEMENT*.

Вся эта глава посвящена конструированию и применению шаблонов. Регулярные выражения Perl – это мощное и невероятно выразительное средство. Поэтому они могут выглядеть устрашающе для того, кто попытается догадаться, что же в целом делает некий длинный шаблон. Но если суметь разбить его на части и знать, как Механизм эти части интерпретирует, можно понять действие любого регулярного выражения. Нередко можно видеть, как сотня строк программы на C или Java выражается на Perl регулярным выражением, состоящим из одной строки. Понять это регулярное выражение может оказаться несколько труднее, чем каждую отдельную строку длинной программы, но, с другой стороны, регулярное выражение, возможно, будет значительно легче понять, чем длинную программу в целом. Просто имейте такие вещи в виду.

Бестиарий регулярных выражений

Прежде чем погрузиться в изучение правил интерпретации регулярных выражений, рассмотрим несколько примеров. Большинство символов в регулярных выражениях просто соответствуют самим себе. Если несколько символов выстроены в ряд, они должны быть найдены, как и можно ожидать, в указанном порядке. Поэтому если записать шаблон для поиска:

```
/Frodo/
```

можно быть уверенным, что поиск будет успешным, только если строка содержит в каком-то месте подстроку "Frodo". (*Подстрока* – это просто часть строки.) Соответствие может находиться в любом месте строки, если только в нем рядом расположены эти пять символов в указанном порядке.

Другие символы не соответствуют самим себе, а некоторым образом «неправильно» ведут себя. Мы называем их *метасимволами*. (Все метасимволы ведут себя неправильно сами по себе, но некоторые из них настолько дурны, что «заражают» плохим поведением и соседние символы.)

Вот эти хулиганы:

```
\ | ( ) [ { ^ $ * + ?
```

Метасимволы на практике очень полезны и имеют в шаблонах особое значение; по ходу дела мы об этих значениях расскажем. Но хотим уверить читателя, что можно найти любой из этих символов и буквально, предварив его символом обратной косой черты. Обратная косая черта, в частности, сама является метасимволом, поэтому для буквального поиска обратной косой черты нужно поместить обратную косую черты перед ней самой: \\.

Как видите, обратная косая черта – это один из тех символов, которые провоцируют неправильное поведение других символов. Отсюда следует, что, заставь мы непослушный символ вести себя неправильно, получим в результате послушание и хорошее поведение – иначе говоря, мы применяем двойное отрицание. Поэтому обратная косая черта перед символом превращает в буквальные лишь символы пунктуации; обратная косая черта перед (обычно хорошо воспитанным) буквенно-цифровым символом производит противоположный эффект: она превращает буквальный символ в нечто особое. Встретив двухсимвольную последовательность вроде этих:

```
\b \D \t \3 \s
```

необходимо знать, что она является *метасимволом*, обозначающим что-то необычное. Например, \b соответствует границе слова, а \t – обычному символу табуляции. Обратите внимание, что табуляция имеет размер в один символ, тогда как граница слова имеет ширину ноль символов, поскольку это место между двумя символами. Поэтому мы называем \b *утверждением нулевой ширины* (*zero-width assertion*). Тем не менее, \t и \b схожи: оба утверждают нечто о конкретном месте в строке. *Утверждая* что-либо в регулярном выражении, мы просто требуем, чтобы это нечто было истинным при соответствии шаблону.

Большинство участков регулярного выражения представляют собой утверждения того или иного рода, при этом обычные символы просто утверждают, что они соответствуют самим себе. Если сказать точнее, они утверждают также, что следующее по очереди соответствие участку регулярного выражения будет располагаться на один символ далее в строке, поэтому мы и говорим, что символ табуляции «имеет ширину в один символ». Некоторые утверждения (такие, как \t) «съедают» в случае соответствия часть строки, в то время как другие (такие, как \b) не делают этого. Но обычно мы используем термин «утверждение» только для утверждений нулевой ширины. Чтобы избежать путаницы, мы будем называть нечто, обладающее шириной, *атомом*. (Если вы физик, можете представлять себе атомы с ненулевой шириной как имеющие массу, в противоположность утверждениям нулевой ширины, которые не имеют массы, как фотоны.)

Мы увидим также, что некоторые метасимволы не являются утверждениями, а выполняют функцию структурирования (подобно тому, как фигурные скобки и точки с запятой определяют структуру обычного кода Perl, но не совершают никаких действий). Эти структурирующие метасимволы в некоторых отношениях являются самыми важными, поскольку при обучении чтению регулярных выражений решающий первый шаг состоит в том, чтобы приучить глаз выхватывать структурирующие метасимволы. После освоения этой премудрости чтение регулярных выражений становится легким, как дуновение ветерка.[1]

[1] Временами, положим, ветерка довольно свежего, но не настолько, чтобы свалить с ног.

Один из таких структурирующих символов – вертикальная черта, обозначающая *перечисление* (*alternation*):

```
/Frodo|Pippin|Merry|Sam/
```

Этот шаблон означает, что обнаружение любой из этих строк делает поиск успешным; об этом рассказывается в разделе «Перечисление» далее в этой главе. А в предшествующем ему разделе «Захват и группировка» мы покажем, как осуществлять *группирование* частей шаблона посредством круглых скобок:

```
/(Frodo|Drogo|Bilbo) Baggins/
```

или даже так:

```
/(Frod|Drog|Bilb)o Baggins/
```

Еще вы познакомитесь с так называемыми *квантификаторами*, которые указывают, сколько раз подряд должно быть найдено то, что им предшествует. Квантификаторы выглядят так:

```
* + ? *? *+ {3} {2,5}
```

Однако они не встречаются в такой изоляции, как здесь. Квантификаторы имеют смысл, только когда прикреплены к атомам, т.е. утверждениям, обладающим шириной.[1] Квантификатор прикрепляется только к предшествующему атому; на человеческом языке это означает, что обычно он указывает количество повторений только одного символа. Чтобы найти в строке три копии подстроки "bar", следующих подряд, нужно сгруппировать отдельные символы "bar" в одной «молекуле» с помощью скобок, вот так:

```
/(bar){3}/
```

Этот шаблон будет соответствовать строке "barbarbar". Если бы вы сказали /bar{3}/, это соответствовало бы "barrr", что служило бы поводом отнести вас к шотландцам, и поставить под сомнение вашу принадлежность к барбарбарианцам. (Некоторые из наших любимых метасимволов можно с уверенностью считать шотландскими.) Подробнее о квантификаторах читайте в разделе «Квантификаторы» далее в этой главе.

Итак, вы посмотрели на некоторых бестий, обитающих в регулярных выражениях, и вам, вероятно, не терпится заняться их укрощением. Однако прежде чем заняться серьезным обсуждением регулярных выражений, надо вернуться немного назад и поговорить об операторах поиска по шаблону, использующих регулярные выражения. (И если по дороге вам случится обнаружить еще несколько зверюшек из регулярных выражений, оставьте приличные чаевые своему гиду.)

---

1 Квантификаторы немного похожи на модификаторы операторов, описанные в главе 4, которые могут прикрепляться только к самостоятельным операторам. Прикрепление квантификатора к утверждению нулевой длины было бы похоже на попытку прикрепить модификатор while к объявлению – и то и другое несет не больше смысла, чем попросить у аптекаря килограмм фотонов. Аптекарь сможет взвесить только атомы и все такое прочее.

Операторы поиска по шаблону

Раз уж мы развиваем метафору зоопарка, можно сказать, что операторы Perl для поиска по шаблону действуют как своего рода крепкие клетки для регулярных выражений. Дело в том, что если выпустить зверюшек из семейства регулярных выражений на просторы языка, Perl может превратиться в настоящие джунгли. Конечно, миру нужны джунгли – в конце концов, они поддерживают биологическое многообразие – но джунгли должны находиться там, где им положено. Аналогично, двигатель комбинаторного многообразия, регулярные выражения, должен оставаться внутри операторов поиска по шаблону, где его место. Там находятся джунгли.

Казалось бы, регулярные выражения и без того мощные, однако операторы m// и s/// дополняют их еще и мощью интерполяции двойных кавычек (аналогичным образом ограниченной). Поскольку шаблоны анализируются как строки в двойных кавычках, действуют все обычные соглашения для двойных кавычек, в том числе интерполяция переменных (если только в качестве ограничителя не используются одинарные кавычки) и специальные символы, обозначаемые с помощью обратной косой черты. (Подробнее – в разделе «Специальные символы» далее в этой главе.) Эти соглашения выполняются до того, как строка интерпретируется в качестве регулярного выражения. (Это одна из редких для Perl ситуаций, когда обработка строки производится более чем в один проход.) Первый проход представляет собой не совсем обычную интерполяцию строки в двойных кавычках, поскольку некоторые части строки следует интерполировать, а некоторые – просто передать анализатору регулярных выражений. Поэтому, например, символ $, сразу за которым следует вертикальная черта, закрывающая круглая скобка или конец строки, будет обрабатываться не как интерполяция переменной, а как обычное утверждение регулярного выражения, означающее конец строки. Поэтому, если сказать:

```
$foo = "bar";
/$foo$/;
```

то на этапе интерполяции Perl будет понимать, что эти два символа $ имеют различное назначение. Поэтому он выполнит интерполяцию $foo и передаст анализатору регулярных выражений следующее:

```
/bar$/;
```

Другим следствием этого двухпроходного анализа является то, что стандартный механизм разбора на лексемы в Perl сначала находит конец регулярного выражения, совсем как если бы искал завершающий ограничитель для обычной строки. Только после того, как найден конец строки (и завершена интерполяция всех переменных), шаблон рассматривается как регулярное выражение. Помимо всего прочего, это означает, что нельзя «скрыть» завершающий ограничитель внутри конструкции регулярного выражения (такой как класс символов или комментарий в регулярном выражении, о которых мы еще расскажем). Perl увидит ограничитель, где бы он ни был, и закончит шаблон в этом месте.

Следует также знать, что интерполяция внедренных в шаблон переменных, значения которых постоянно меняются, замедляет работу механизма сопоставления с шаблоном, если ему приходится заново компилировать шаблон. См. раздел «Интерполяция переменных» далее в этой главе. Вы можете насильственно подавить

повторную компиляцию с помощью модификатора /o (old – старый), но обычно лучше вынести изменяющиеся части за пределы шаблона посредством конструкции qr//, чтобы повторной компиляции подвергалась только та часть, которую действительно необходимо скомпилировать повторно.

Оператор транслитерации[1] tr/// не производит интерполяцию переменных и даже не задействует регулярные выражения! (На самом деле ему, возможно, вообще не место в этой главе, но лучшего места для него мы не нашли.) Однако у него есть одна общая черта с m// и s///: он выполняет привязку к переменным с помощью операторов =~ и !~.

Операторы =~ и !~, описанные в главе 3, привязывают скалярное выражение из своей левой части к одному из трех операторов типа кавычек, находящихся в правой части: m// для поиска по шаблону, s/// для подстановки некоторой строки вместо подстроки, сопоставленной с шаблоном, и tr/// (или синоним – y///) для транслитерации (замены) одного набора символов другим. (Можно также записывать m// как //, без m, если в качестве ограничителя выступает косая черта.) Если в правой части оператора =~ или !~ находится нечто отличное от этих трех операторов, оно все равно рассматривается как операция поиска m//, но при этом не остается места для каких-либо замыкающих модификаторов (см. далее раздел «Модификаторы шаблонов»), и вам придется обрабатывать свои собственные кавычки:

```
say "соответствует" if $somestring =~ $somepattern;
```

На самом деле нет оснований, чтобы не написать это явно:

```
say "соответствует" if $somestring =~ m/$somepattern/;
```

В операциях сопоставления операторы =~ и !~ иногда читаются, как «подходит» (matches) и «не подходит», соответственно (хотя пара «содержит»/«не содержит» создавала бы не так много путаницы).

Регулярные выражения в Perl фигурируют еще в паре мест, помимо операторов m// и s///. Первый аргумент функции split является особым оператор сопоставления, определяющим, что *не должно* возвращаться при разбиении строки на несколько подстрок. Описание и примеры применения split даются в главе 27. Оператор qr// («quote regex» – заключить в кавычки регулярное выражение) тоже задает шаблон посредством регулярного выражения, но выполняет сопоставление (в отличие от m//, который это делает). Вместо этого qr// возвращает скомпилированную форму регулярного выражения, которую можно использовать в дальнейшем. Дополнительные сведения можно найти в разделе «Интерполяция переменных» данной главы.

Один из операторов m//, s/// или tr/// можно применить для конкретной строки посредством оператора привязки =~ (который, в сущности, является не оператором, а некоторым «тематизатором», рассуждая в терминах лингвистики). Вот несколько примеров:

```
$haystack =~ m/needle/                # поиск по простому шаблону
$haystack =~  /needle/                # то же
```

[1] transliteration operator – обычно переводят как «оператор замены», но термин «транслитерация» в данном случае точнее отражает суть происходящего, поскольку указывает на побуквенную замену. – *Прим. перев.*

```
$italiano =~ s/butter/olive oil/      # полезная для здоровья замена

$rotate13 =~ tr/a-zA-Z/n-za-mN-ZA-M/  # простое (для раскрытия) шифрование
```

В отсутствие оператора привязки в качестве «темы» неявным образом используется переменная $_:

```
/new life/ and                  # искать в $_ и (если найдена)
    /new civilizations/         #    смело снова искать в $_

s/sugar/aspartame/              # заменить "сахар" на "заменитель" в $_

tr/ATCG/TAGC/                   # дополнить цепочку ДНК в $_
```

Поскольку s/// и tr/// изменяют скалярные значения, к которым применяются, их можно использовать только с допустимыми левыми значениями:[1]

```
"onshore" =~ s/on/off/;         # НЕВЕРНО: ошибка этапа компиляции
```

Однако m// работает с результатом любого скалярного выражения:

```
if ((lc $magic_hat->fetch_contents->as_string) =~ /rabbit/) {
    say "Э-эй, док, какие дела?";
}
else {
    say "Этот фокус никогда не срабатывает!";
}
```

Но следует проявлять некоторую осторожность, поскольку =~ и !~ имеют достаточно высокий приоритет: в предшествующем примере пришлось заключить левый терм в круглые скобки.[2] Оператор привязки !~ действует подобно =~, но меняет логический результат операции на противоположный:

```
if ($song !~ /words/) {
    say qq/Похоже, что "$song" является песней без слов./
}
```

Поскольку m//, s/// и tr/// являются операторами цитирования, мы можем выбрать собственные ограничители в качестве кавычек. Они действуют так же, как операторы q//, qq//, qr// и qw// (см. раздел «Выберите собственные кавычки» в главе 2).

```
$path =~ s#/tmp#/var/tmp/scratch#;

if ($dir =~ m[/bin]) {
    say "Пожалуйста, без каталогов исполняемых программ.";
}
```

Если в s/// или tr/// используются парные ограничители, и пара в первой части является одной из традиционных (угловые, круглые, квадратные или фигурные), то для второй части можно выбрать ограничители, отличающиеся от выбранных для первой:

---

1 Если не используется модификатор /r, возвращающий результат замены как r-значение.

2 В отсутствие скобок имеющая более низкий приоритет функция lc применяется ко всему шаблону поиска, а не только к вызову метода объекта «волшебный цилиндр».

```
s(egg)<larva>;
s{larva}{pupa};
s[pupa]/imago/;
```

Перед открывающими ограничителями допустимы пробельные символы:

```
s (egg) <larva>;
s {larva} {pupa};
s [pupa] /imago/;
```

Всякий раз, когда поиск по шаблону оказывается успешным (в том числе при замене по шаблону), переменные $`, $& и $' устанавливаются равными тексту слева от найденного соответствия, всему соответствию и тексту справа от соответствия. Это удобно для разделения строк на части:

```
"hot cross buns" =~ /cross/;
say "Найдено: <$`> $& <$'>";        # Найдено: <hot > cross < buns>
say "Слева: <$`>";                  # Слева: <hot >
say "Соответствие: <$&>";           # Соответствие: <cross>
say "Справа: <$'>";                 # Справа: < buns>
```

Детализацию и эффективность можно увеличить при помощи скобок, позволяющих захватывать отдельные участки того, что требуется сохранить. Каждая пара круглых скобок захватывает подстроку, соответствующую *подшаблону*, заключенному в эти скобки. Пары скобок нумеруются слева направо в соответствии с позициями левых скобок; подстроки, соответствующие этим подшаблонам, после окончания поиска доступны как пронумерованные переменные $1, $2, $3 и так далее:[1]

```
$_ = "Бильбо Бэггинс родился 22 сентября";
/(.*) родился (.*)/;
say "Имя: $1";
say "Дата: $2";
```

Переменные $`, $&, $', а также нумерованные переменные являются глобальными, неявно локализованными в охватывающей динамической области видимости. Их значения сохраняются до очередного успешного поиска по шаблону или до конца текущей области видимости – что наступит раньше. Подробнее об этом потом, в другой области видимости.

Когда только Perl обнаруживает, что где-то в программе использована одна из переменных, $`, $& или $', то начинает создавать их при каждом поиске по шаблону. Это несколько замедляет выполнение программы. Аналогичный механизм используется в Perl для создания переменных $1, $2 и так далее, поэтому приходится платить и за каждый шаблон, содержащий сохраняющие скобки. (В разделе «Несохраняющая группировка» мы расскажем, как избежать издержек захвата соответствий при сохранении режима группирования.) Но если ни разу не использовать $`, $& или $', то шаблоны *без* сохраняющих скобок штрафом не облагаются. Поэтому обычно лучше избегать применения $`, $& и $', если есть такая возможность, особенно в библиотечных модулях. Но если вы хоть раз их использовали (а в некоторых алгоритмах это действительно удобно), то можете дальше уже не особенно стесняться, поскольку за все заплачено. В последних версиях Perl переменная $& требует меньших расходов, чем две другие.

---

[1] Переменная $0 в их число не входит, поскольку хранит имя программы.

Более удачная альтернатива заключается в использовании модификатора /p, обсуждаемого ниже. Он сохраняет соответствия так, что переменные ${^PREMATCH}, ${^MATCH} и ${^POSTMATCH} содержат то, что могли бы содержать переменные $`, $& и $', но штрафуется при этом не вся программа, а только текущее регулярное выражение.

Модификаторы шаблонов

Мы обсудим отдельные операторы поиска по шаблону буквально через мгновение, но сначала хотелось бы рассказать еще об одном общем знаменателе: *модификаторах*.

Сразу за последним ограничителем оператора m//, s///, qr//, y/// или tr/// можно поместить один или несколько необязательных односимвольных модификаторов, в произвольном порядке. Для большей ясности модификаторы обычно пишутся как «модификатор /i» и произносятся «модификатор слеш ай», даже если в качестве закрывающего ограничителя используется не косая черта. (Иногда говорят «флаг» или «ключ», имея в виду «модификатор»; это тоже приемлемо.)

Некоторые модификаторы изменяют режим действия отдельного оператора, подробно мы опишем их позже. Другие изменяют порядок интерпретации регулярного выражения, и мы поговорим о них сейчас. Операторы m//, s/// и qr//[1] – все принимают модификаторы, перечисленные в табл. 5.1, после своего конечного ограничителя:

Таблица 5.1. Модификаторы регулярных выражений

Модификатор	Значение
/i	Игнорировать различия в регистре символов
/s	Позволить символу . соответствовать переводу строки
/m	Позволить символам ^ и $ соответствовать позиции рядом с внедренным символом /n
/x	Игнорировать (большинство) пробельных символов и разрешить комментарии в шаблоне
/o	Компилировать шаблон только один раз
/p	Поддерживать переменные ${^PREMATCH}, ${^MATCH} и ${^POSTMATCH}
/d	Двойной режим работы с набором символов ASCII-Юникод (прежний режим по умолчанию)
/a	Режим работы с набором символов ASCII
/u	Режим работы с набором символов Юникода (новый режим по умолчанию)
/l	Режим работы с набором символов, определяемым региональными настройками времени выполнения (режим по умолчанию при использовании прагмы use locale)

Модификатор /i указывает, что соответствие символов устанавливается независимо от их регистра, т.е. поиск выполняется без учета регистра символов. Этот

[1] Оператор tr/// не позволяет использовать регулярные выражения, поэтому к нему неприменимы эти модификаторы.

процесс также называется *сверткой регистра* (*casefolding*). Это означает, что шаблону будут соответствовать не только символы верхнего и нижнего регистров, но и заглавного регистра (в английском языке такого регистра нет). Поиск без учета регистра символов также необходим, когда символы имеют несколько вариантов в одном и том же регистре, как, например, символ «сигма» в греческом алфавите: заглавной букве «Σ» обычно соответствует строчная «σ», которая в конце слова превращается в «ς». Например, греческое слово *Σίσυφος* («Сизиф») содержит все три символа «сигма».

Поскольку сопоставление без учета регистра символов выполняется посимвольно и не зависит от языка,[1] соответствие может обнаруживаться в строках, имеющих неправильное начертание в том или ином языке. Например, /perl/i совпадет не только со строкой "perl", но также найдет соответствие в строках "proPErly" и "perLiter", хотя таких слова нет в английском языке. Аналогично и шаблон на /σίσυφος/i на греческом может совпасть не только со словами «ΣΊΣΥΦΟΣ» и «Σίσυφος», но также с неправильно записанным словом «ςίσυφοσ», в котором первая и вторая сигмы в нижнем регистре поменялись местами.

То есть, даже при том, что мы идентифицировали наши строки, как текст на английском или греческом языке, Perl совершенно не в курсе этого. Он просто выполняет поиск без учета регистра символов, игнорируя любые особенности языка. Поскольку свертка всех возможных регистров одной и той же буквы приводит к одному и тому же результату, все они обнаруживаются шаблоном.

Так как Perl поддерживает только 8-битные наборы символов в региональных настройках, свертка регистров символов с кодовыми пунктами ниже 256 выполняется с применением текущей таблицы символов, а символов с большими значениями кодов используются правила Юникода. Сопоставление без учета регистра символов в рамках региональных настроек не может преодолеть границу 255/256, при этом могут накладываться и другие ограничения.

Модификаторы /s и /m не связаны с какими-либо извращениями. Они изменяют способ, которым Perl осуществляет поиск в строке, содержащей символы перевода строки. Важно не то, присутствует ли действительно символ \n в анализируемой строке, а следует ли Perl *считать*, что последовательность символов содержит одну строку (/s) или несколько (/m), поскольку некоторые метасимволы поразному действуют – в зависимости от того, предполагается ли их использование в режиме, ориентированном на строки, или нет.

Обычно метасимвол "." (точка) соответствует любому символу, *кроме* символа перевода строки, потому что традиционно он предназначается для поиска символов в одной строке текста (line). Однако при наличии модификатора /s метасимвол "." может соответствовать и символу перевода строки, поскольку этот модификатор сообщает Perl о необходимости игнорировать то обстоятельство, что последовательность символов может содержать символы перевода строки. Если необходимо предотвратить соответствие метасимвола "." символу перевода строки при использовании модификатора /s, просто используйте класс символов \N, который означает то же, что и [^\n], но проще в наборе с клавиатуры.

---

1 Или *почти* не зависит. Мы не хотели бы обсуждать проблему с символом «İ» в турецком алфавите, поэтому не будем этого делать.

Напротив, модификатор /m изменяет интерпретацию метасимволов ^ и $, позволяя им соответствовать позициям рядом с символами перевода строки внутри текста, а не только по концам строки (модификатор /m может отключать оптимизации, предполагающие поиск в пределах одной строки текста, а не по всему тексту, поэтому не применяйте его бездумно). Примеры приводятся в разделе «Позиции» далее в этой главе.

Модификатор /p включает поддержку сохранения текста соответствия в специальной переменной ${^MATCH}; текста, предшествующего соответствию, в переменной ${^PREMATCH}; и текста, следующего за соответствием, в переменной ${^POSTMATCH}.

Ныне почти окончательно устаревший модификатор /o управляет перекомпиляцией шаблона. В настоящее время длина шаблона должна превышать 10 Кбайт, чтобы выгода от применения этого модификатора стала хоть сколько-нибудь заметной, поэтому данный модификатор можно считать пережитком прошлого. Тем не менее, вы можете столкнуться с ним в старом программном коде, поэтому посмотрим, как он действует.

За исключением случаев, когда ограничителями служат одинарные кавычки (m'*PATTERN*', s'*PATTERN*'*REPLACEMENT*' или qr'*PATTERN*'), все переменные в шаблоне интерполируются при каждом вычислении оператора шаблона. В худшем случае выполняется полная перекомпиляция шаблона, а в лучшем дело ограничивается сравнением строки с целью определить необходимость перекомпиляции. Чтобы шаблон компилировался один и только один раз, используйте модификатор /o. Это позволяет избежать дорогостоящей перекомпиляции на этапе выполнения и полезно, если интерполируемое значение во время выполнения не меняется. Однако модификатор /o – это еще и обещание не изменять переменные, входящие в шаблон. Если же они все-таки изменятся, Perl не обратит на это никакого внимания. Лучшее управление перекомпиляцией достигается при использовании оператора цитирования регулярных выражений, qr//. Подробности читайте далее в этой главе, в разделе «Интерполяция переменных».

Модификатор /x является *экс*прессивным: он позволяет *экс*плуатировать пробельные символы и пояснительные комментарии в целях улучшения понятности вашего шаблона и даже *экс*пансии шаблона через границы символов перевода строки.

Это означает, что /x изменяет значение пробельных символов (и символа #): вместо того чтобы при поиске «притвориться» обычными символами, они превращаются в метасимволы, которые, как ни странно, ведут себя в результате так, как и должны пробельные символы (и символ комментария). Следовательно, благодаря /x пробелы, символы табуляции и перевода строки используются для форматирования, как в обычном коде Perl. Также благодаря этому модификатору за символом #, обычно не являющимся в шаблоне каким-либо особенным, можно размещать комментарии, продолжающиеся до конца текущей строчки (line) в шаблоне.[1] Если действительно потребуется найти пробельный символ (или символ #), придется поместить его в класс символов, либо экранировать его обратной косой чертой, либо выразить его соответствующим восьмеричным или шестнадцатеричным

[1] Следите, чтобы в комментарий не попал ограничитель шаблона, ведь следуя своему правилу «сначала найти конец», Perl не сможет определить, что вы не собирались закончить шаблон в этом месте.

кодом. (Но обычно поиск пробельных символов осуществляется утверждениями \s* или \s+, поэтому на практике такая ситуация возникает редко.)

Совместное применение этих функций значительно способствует превращению обычных регулярных выражений в доступный для чтения язык. В соответствии с духом TMTOWTDI теперь есть больше одного способа записи данного регулярного выражения. На самом деле их больше двух:

```
m/\w+:(\s+\w+)\s*\d+/;       # Слово, двоеточие, пробел, слово, пробел, цифры

m/\w+: (\s+ \w+) \s* \d+/x; # Слово, двоеточие, пробел, слово, пробел, цифры

m{
    \w+:                     # Найти слово и двоеточие.
    (                        # (начало сохраняющей группы)
        \s+                  # Найти один или несколько пробелов.
        \w+                  # Найти еще одно слово
    )                        # (конец сохраняющей группы)
    \s*                      # Найти ноль или несколько пробелов.
    \d+                      # Найти несколько цифр
}x;
```

Эти новые метасимволы мы опишем ниже. (Данный раздел мы хотели посвятить модификаторам шаблонов, но, впав в *экстаз* по поводу /x, не смогли удержаться в заданных рамках.) Вот регулярное выражение, отыскивающее повторяющиеся слова в абзацах и украденное прямо из «Perl. Библиотека программиста». В нем используются модификаторы /x и /i, а также модификатор /g, который будет описан ниже.

```
# Найти в абзацах повторяющиеся слова, возможно, пересекая границы строчек
#   Применяется /x для пробелов и комментариев, /i для поиска обоих 'is'
#   в "Is is this ok?", и /g для поиска всех дубликатов
$/ = "";                       # режим "paragrep"
while (<>) {
    while ( m{
                \b             # начать с границы слова
                (\w\S+)        # найти похожий на слово фрагмент
                (
                    \s+        # отделенный каким-нибудь пробелом,
                    \1         # и еще раз этот же фрагмент.
                ) +            # Повторять, сколько возможно,
                \b             # до следующей границы слова
            }xig
        )
    {
        say "повтор слова '$1' в абзаце $.";
    }
}
```

Если применить этот код к данной главе, можно получить что-то вроде:

```
повтор слова 'that' в абзаце 150
```

В данном случае мы знаем, что это повторение является преднамеренным.

Модификатор /u включает использование семантики Юникода при сопоставлении. Он устанавливается автоматически, если шаблон представлен в кодировке

UTF-8 или был скомпилирован в области действия прагмы use feature "unicode_strings" (а также не был скомпилирован в области действия старой прагмы use locale или use bytes, каждая из которых не рекомендуется к использованию).

В области действия модификатора /u символы с кодовыми пунктами 128–255 (т.е. между 128 и 255 включительно) интерпретируются как символы из набора ISO-8859-1 (Latin-1), и соответствуют символам с теми же кодами Юникода. Без модификатора /u метасимвол \w в строках, имеющих кодировку, отличную от UTF-8, точно соответствует символам [A-Za-z0-9_] и ничему больше. С модификатором /u метасимвол \w в строках, имеющих кодировку, отличную от UTF-8, также соответствует всем «буквам» Latin-1 с кодовыми пунктами в диапазоне 128–255, а именно: символу MICRO SIGN µ, двум индикаторам единиц измерения, ª и º, и 62 латинским буквам. (В строках UTF-8 метасимвол \w также соответствует всем этим символам.)

Модификатор /a изменяет действие метасимволов \d, \s, \w и POSIX-классов символов так, чтобы они соответствовали только символам из набора ASCII.[1] Эти метасимволы обычно соответствуют кодам Юникода, не обязательно входящим в диапазон ASCII. Однако в области действия модификатора /a метасимвол \d соответствует только десяти цифрам ASCII, от «0» до «9», \s соответствует только пяти пробельным символам ASCII [\f\n\r\t], а \w соответствует только 63 «буквам» ASCII [A-Za-z0-9_]. (Это касается также метасимволов \b и \B, поскольку они определяются через метасимвол \w.) Точно так же все POSIX-классы вроде [[:print:]] соответствуют символам ASCII только в области действия модификатора /a.

Кое-чем модификатор /a намного больше походит на /u, чем можно было бы подумать: он не гарантирует, что символам ASCII будут соответствовать только символы ASCII. Например, согласно правилам свертки регистра в Юникоде, все символы – «S», «s» и «ſ» (U+017F LATIN SMALL LETTER LONG S) – будут соответствовать друг другу при сопоставлении без учета регистра символов, так же как и символы «K», «k» и U+212A KELVIN SIGN, «K». Запретить такое причудливое поведение механизма свертки регистра в Юникоде можно, продублировав модификатор: /aa.

Модификатор /l вынуждает механизм сопоставления с шаблоном использовать текущие региональные настройки. Под «текущими региональными настройками» здесь подразумеваются настройки, действующие на момент сопоставления, а не действовавшие на момент компиляции шаблона. В системах, поддерживающих такую возможность, текущие региональные настройки могут изменяться с помощью функции setlocale из модуля POSIX. Этот модификатор действует автоматически для шаблонов, скомпилированных в области действия прагмы "use locale".

Perl поддерживает только региональные настройки с наборами однобайтных символов. Это означает, что кодовые пункты со значениями выше 255 интерпретируются как символы Юникода, независимо от текущих региональных настроек. В соответствии с правилами Юникода, при поиске без учета регистра символов механизм сопоставления может перешагивать границу однобайтных символов между значениями кодовых пунктов 255 и 256, но модификатор /l из необходимости запрещает такое поведение.

---

[1] Когда мы говорим о наборе символов ASCII, все, кто по-прежнему используют набор EBCDIC, должны в уме делать соответствующую подстановку во время чтения. Подробнее о наборе EBCDIC рассказывается в электронной документации Perl.

Это обусловлено тем, что в некоторых национальных алфавитах символам назначаются кодовые пункты, не совпадающие с кодами символов в Юникоде (исключение составляет набор ISO-8859-1). По этой причине, например, национальный символ с кодом 255 не совпадает при поиске без учета регистра с символом, имеющим код 376, U+0178 LATIN CAPITAL LETTER Y WITH DIAERESIS (Ÿ), так как код 255 может *не* соответствовать символу U+00FF LATIN SMALL LETTER Y WITH DIAERESIS (ÿ) в текущем национальном алфавите. В Perl отсутствует механизм, с помощью которого можно было бы определить, существует ли требуемый символ в текущем наборе символов, а уж тем более, какой у него может быть код.

Модификатор /u включается по умолчанию, если вы явным образом предписали набора особенностей Perl версии v5.14. В противном случае существующий программный код будет действовать как прежде, как если бы вы использовали модификатор /d в каждом шаблоне (или /l, в области действия use locale). Это гарантирует обратную совместимость, а также обеспечивает более элегантный способ реализации функций в будущем. Традиционно механизм сопоставления с шаблонами в языке Perl демонстрирует двойственное поведение, откуда и взялось название модификатора /d (dual – двойственный), которое можно также перевести, как «it depends» (в зависимости от обстоятельств). В области действия модификатора /d правила сопоставления определяются набором символов, используемым платформой, если нет каких-либо указаний на необходимость применять правила Юникода. В число последних входят:

- Внутренней кодировкой целевой строки или самого шаблона является UTF-8;
- присутствуют символы с кодами выше 255;
- используются спецификаторы свойств вида \p{*СВОЙСТВО*} или \P{*СВОЙСТВО*};
- используются именованные символы, псевдонимы или последовательности вида \N{*NAME*}, или кодовые пункты вида \N{U+*HEXDIGITS*}.

В отсутствие каких-либо объявлений, принудительно устанавливающих семантику модификаторов /u, /a и /l, по умолчанию используется двойной (dual) режим, /d. Шаблоны с модификатором /d могут проявлять или не проявлять поведение, свойственное режиму работы с Юникодом. Традиционно такое смешивание семантик ASCII и Юникода служило источником бесконечной путаницы, поэтому в области действия прагмы use v5.14 данный модификатор больше не действует по умолчанию. Однако вы можете явным образом включить режим Юникода. Поддержку строк Юникода можно включить любым из следующих способов:

```
use feature "unicode_strings";
use feature ":5.14";
use v5.14;
use 5.14.0;
```

Поддержку строк Юникода можно также включить с помощью ключей командной строки Perl, соответствующих четырем прагмам, перечисленным выше:

```
% perl -Mfeature=unicode_strings дополнительные аргументы
% perl -Mfeature=:5.14 дополнительные аргументы
% perl -M5.014 дополнительные аргументы
% perl -M5.14.0 дополнительные аргументы
```

Поскольку ключ командной строки -E подразумевает использование набора особенностей текущей версии Perl, он также включает поддержку строк Юникода (в v5.14+):

```
% perl -E              программный код для выполнения
```

Как для большинства прагм, имеется возможность выключать отдельные особенности в отдельных областях видимости, так что прагма

```
no feature "unicode_strings",
```

отключит семантику Юникода в охватывающей лексической области видимости.

Чтобы упростить управление поведением регулярных выражений, не прибегая каждый раз к использованию модификаторов, можно задействовать прагму re для управления флагами по умолчанию в лексической области видимости.

```
# определить модификаторы по умолчанию для всех шаблонов
use re "/msx"; # эти модификаторы будут автоматически добавляться ко всем
               # шаблонам в данной области видимости

# отменить некоторые из модификаторов для вложенной области видимости
{
    no re "/ms"; # эти модификаторы будут автоматически убираться из шаблонов
                 # в данной области видимости
    ...
}
```

Это особенно удобно, когда речь идет о модификаторах, относящихся к набору символов:

```
use re "/u";  # Режим Юникода
use re "/d";  # двойственный режим, ASCII-Юникод
use re "/l";  # режим использования набора 8-битных национальных символов
use re "/a";  # режим ASCII, плюс механизм свертки регистра из Юникода
use re "/aa"; # жесткий режим ASCII, без механизма свертки регистра из Юникода
```

Благодаря этим объявлениям вам не придется вновь и вновь повторять одни и те же модификаторы, чтобы обеспечить последовательное применение нужной семантики, или даже последовательное применение неправильной семантики.

Оператор m// (поиск)

```
m/PATTERN/модификаторы
 /PATTERN/модификаторы
 ?PATTERN?модификаторы (устаревшая форма)

EXPR =~ m/PATTERN/модификаторы
EXPR =~ /PATTERN/модификаторы
EXPR =~ ?PATTERN?модификаторы (устаревшая форма)
```

Оператор m// ищет в строке скаляра *EXPR* соответствие шаблону *PATTERN*. Если ограничителем служит символ / или ?, то присутствие начальной m не обязательно. Символы ? и ' имеют специальное значение в качестве ограничителей: первый служит для поиска единственного соответствия, а второй подавляет интерполяцию переменных и семь escape-последовательностей трансляции (\U и прочих, они описаны далее).

Если результатом вычисления *PATTERN* оказывается пустая строка – например, потому, что вы указали пустой шаблон, или потому, что интерполируемая переменная содержала пустую строку, то для поиска используется последнее успешно обработанное регулярное выражение, не скрытое во внутреннем блоке (или в split, grep или map).

В скалярном контексте оператор возвращает в случае успешного выполнения значение «истина» (1), а в противном случае значение «ложь» (""). В логическом контексте обычно можно видеть такую форму:

```
if ($shire =~ m/Baggins/) { ... }  # искать Baggins в $shire
if ($shire =~ /Baggins/)  { ... }  # искать Baggins в $shire

if ( m#Baggins# )         { ... }  # искать прямо в $_
if ( /Baggins/ )          { ... }  # искать прямо в $_
```

В списочном контексте оператор m// возвращает список подстрок, соответствующих сохраняющим скобкам в шаблоне (т.е. $1, $2, $3 и т.д.), как описывается далее, в разделе «Захват и группировка». Нумерованным переменным присваиваются значения даже в списочном контексте. Если поиск в списочном контексте не дал результатов, возвращается пустой список. А если успешен, но сохраняющих скобок в шаблоне нет (как и модификатора /g), возвращается список со значением (1). Поскольку при безрезультатном поиске возвращается пустой список, этот формат m// можно применять в логическом контексте, но только в косвенном виде, через присваивание списку:

```
if (($key,$value) = /(\w+): (.*)/) { ... }
```

Допустимые для m// (в любом его обличье) модификаторы перечислены в табл. 5.2.

Таблица 5.2. Модификаторы m//

Модификатор	Значение
/i	Игнорировать различия в регистре символов
/m	Позволить символам ^ и $ соответствовать позиции рядом с внедренным символом /n
/s	Позволить символу . соответствовать переводу строки
/x	Игнорировать (большинство) пробельных символов и разрешить комментарии в шаблоне
/o	Компилировать шаблон только один раз
/p	Сохранять найденные соответствия
/d	Двойной режим работы с набором символов ASCII-Юникод (прежний режим по умолчанию)
/u	Режим работы с набором символов Юникода (новый режим по умолчанию)
/a	Режим работы с набором символов ASCII
/l	Режим работы с набором символов, определяемым региональными настройками времени выполнения (режим по умолчанию при использовании прагмы use locale)
/g	Глобальный поиск всех соответствий
/cg	Разрешить продолжение поиска после неудачи поиска /g

Большинство из этих модификаторов относятся к регулярному выражению и были описаны выше. Последние два меняют режим работы самого оператора поиска. Модификатор /g предписывает глобальный поиск, т.е. поиск всех соответствий шаблону, содержащихся в строке. Однако его действие зависит от контекста. В списочном контексте m//g возвращает список всех найденных соответствий. Найдем, например, все места, где упоминается "perl", "Perl", "PERL" и тому подобные варианты:

```
if (@perls = $paragraph =~ /perl/gi) {
    printf "Perl упомянут %d раз.\n", scalar @perls;
}
```

Если в шаблоне с модификатором /g нет сохраняющих скобок, возвращаются полные соответствия. Если сохраняющие скобки есть, возвращаются только захваченные строки. Пусть имеется такая строка:

```
$string = "password=xyzzy verbose=9 score=0";
```

Пусть также требуется использовать ее для инициализации хеша, как показано ниже:

```
%hash = (password => "xyzzy", verbose => 9, score => 0);
```

Мешает, конечно, то обстоятельство, что исходно мы имеем строку вместо списка. Чтобы получить соответствующий список, можно применить оператор m//g в списочном контексте и захватить все пары ключ/значение, имеющиеся в строке:

```
%hash = $string =~ /(\w+)=(\w+)/g;
```

Последовательность (\w+) захватывает слово из алфавитно-цифровых символов. См. раздел «Захват и группировка» далее в этой главе.

В скалярном контексте модификатор /g предписывает *поступательный поиск*, что заставляет Perl искать следующее соответствие для той же переменной, начиная с позиции, следующей сразу за последним найденным соответствием. Утверждение \G представляет эту позицию в строке (см. описание \G далее в этой главе, в разделе «Позиции»). Если помимо /g действует также модификатор /c («continue»), то, когда /g заканчивает работу, последний безрезультатный поиск не сбрасывает указатель позиции.

Если в качестве ограничителя выступает ?, как в *m?PATTERN?* (или просто *?PATTERN?*, но форма без *m* является устаревшей), поиск выполняется, как обычный */PATTERN/*, но при этом мы ищем только одно соответствие между вызовами оператора reset. Такой прием может дать удобный способ оптимизации, если во время прогона программы требуется найти не все соответствия шаблону, а только первое. Оператор осуществляет поиск при каждом вызове, пока, наконец, что-нибудь не найдет, после чего выключается и будет возвращать «ложь», пока не будет повторно включен с помощью явного вызова reset. Работу по отслеживанию состояния поиска Perl берет на себя.

Оператор m?? полезнее всего в ситуации, когда обычный поиск по шаблону находит последнее, а не первое соответствие:

```
open(DICT, "/usr/dict/words") or die "Невозможно открыть words: $!\n";
while (<DICT>) {
    $first = $1 if m? (^ neur .* )?x;
```

```
    $last  = $1 if m/ (^ neur .* )/x,
}
say $first:      # выведет "neurad"
say $last;       # выведет "neurypnology"
```

Оператор reset сбрасывает только экземпляры ??, скомпилированные в том же пакете, что и вызов reset. Высказывание m?? эквивалентно высказыванию ??.

Оператор s/// (подстановка)

```
s/PATTERN/REPLACEMENT/              модификаторы

 LVALUE =~ s/PATTERN/REPLACEMENT/   модификаторы
 RVALUE =~ s/PATTERN/REPLACEMENT/r  модификаторы
```

Этот оператор ищет в строке соответствие шаблону *PATTERN* и, если оно найдено, заменяет найденную подстроку текстом *REPLACEMENT*. Если *PATTERN* – пустая строка, используется последнее успешно обработанное регулярное выражение.

```
$lotr = $hobbit;           # Просто копируем Хоббита
$lotr =~ s/Bilbo/Frodo/g;  # и легко пишем продолжение.
```

При наличии модификатора /r оператор s/// возвращает строку результата, а целевая строка слева остается неизмененной. Без модификатора /r оператором s/// возвращается значение (одинаковое в скалярном и списковом контекстах), равное количеству выполненных подстановок (оно может быть больше одного при использовании модификатора /g, описанного выше). В случае неудачи, поскольку количество замен равно нулю, возвращается значение «ложь» (""), численный эквивалент 0.[1]

```
if ($lotr =~ s/Bilbo/Frodo/) { say "Продолжение успешно написано" }
$change_count = $lotr =~ s/Bilbo/Frodo/g;
```

Обычно любое соответствие шаблону *PATTERN* теряется при замене, однако имеется возможность «сохранить» соответствия, включив в шаблон метасимвол \K:

```
$tales_of_Rohan =~ s/Éo\Kmer/wyn/g  # переписать историю
```

Часть оператора, определяющая заменяющий текст (*REPLACEMENT*), рассматривается как строка в двойных кавычках. В строке замены можно использовать любые из описанных выше переменных с динамической областью видимости ($`, $&, $', $1, $2 и другие), относящихся к шаблонам, а также любые другие связанные с двойными кавычками приемы, которые вам вздумается применить. Вот, например, как найти все строки "revision", "version", "release" и заменить каждую из них эквивалентом из заглавных букв с помощью управляющего символа \u в части подстановки:

```
s/revision|version|release/\u$&/g; # Используйте | для обозначения
                                   # "или" в шаблоне
```

1 Как и при использовании оператора m//, и многих других более привычных операторов, описанных в главе 3, это – особое «ложное» значение, которое безопасно использовать в качестве числа. Это обусловлено тем, что, в отличие от обычной пустой строки, данное значение не приводит к предупреждениям о неявном преобразовании строки в число.

В контексте двойных кавычек разыменовываются все скалярные переменные, а не только эти необычные. Допустим, у нас есть хеш %Names, где хранятся соответствия номеров версий внутренним названиям проектов; например, $Names{"3.0"} содержит кодовое имя "Isengard". Можно посредством s/// найти номера версий и заменить их соответствующими названиями проектов:

```
s/version ([0-9.]+)/the $Names{$1} release/g;
```

В строке подстановки $1 возвращает то, что сохранила первая (и единственная) пара скобок. (Можно также использовать формат \1, как в шаблоне, но в строке подстановки такой стиль не приветствуется. В обычной строке, заключенной в двойные кавычки, \1 означает Control-A.)

Интерполяция переменных происходит в обеих строках, *PATTERN* и *REPLACEMENT*, но *PATTERN* интерполируется всякий раз, когда оператор s/// выполняется в целом, а *REPLACEMENT* интерполируется каждый раз, когда найдено соответствие шаблону. (*PATTERN* может находить несколько соответствий при каждом прогоне, если указан модификатор /g.)

Как и ранее, большинство модификаторов в табл. 5.3 изменяет действие регулярного выражения; они те же, что в m// и qr//. Последние три модификатора изменяют сам оператор подстановки.

Таблица 5.3. Модификаторы s///

Модификатор	Значение
/i	Игнорировать различия в регистре символов
/m	Позволить символам ^ и $ соответствовать позиции рядом с внедренным символом /n
/s	Позволить символу . соответствовать переводу строки
/x	Игнорировать (большинство) пробельных символов и разрешить комментарии в шаблоне
/o	Компилировать шаблон только один раз
/p	Сохранять найденные соответствия
/d	Двойной режим работы с набором символов ASCII-Юникод (прежний режим по умолчанию)
/u	Режим работы с набором символов Юникода (новый режим по умолчанию)
/a	Режим работы с набором символов ASCII
/l	Режим работы с набором символов, определяемым региональными настройками времени выполнения (режим по умолчанию при использовании прагмы use locale)
/g	Глобальный поиск всех соответствий
/r	Вернуть результат подстановки, не изменять исходную строку
/e	Вычислять правую часть как выражение

Модификатор /g используется с оператором s/// для замены всех соответствий *PATTERN* значением *REPLACEMENT*, а не только первого найденного. Оператор s///g действует как глобальный поиск с заменой, внося все изменения одновременно, даже в скалярном контексте (в отличие от m//g, выполняющего поступательный поиск).

Модификатор /r (неразрушающее воздействие) выполняет подстановку в новой копии исходной строки, которая уже не должна быть переменной, и возвращает копию, независимо от того, была ли выполнена подстановка – исходная строка не изменяется ни при каких обстоятельствах:

```
say "Deagol's ring!" =~ s/D/Sm/r; # выведет "Smeagol's ring!"
```

Копия всегда будет простой строкой, даже если на вход оператора подать объект или связанную переменную. Этот модификатор появился в Perl версии v5.14.

Модификатор /e рассматривает *REPLACEMENT* как фрагмент кода Perl, а не как интерполируемую строку. Результат выполнения этого кода выступает в качестве строки подстановки. Например, s/([0-9]+)/sprintf("%#x", $1)/ge преобразует все числа в шестнадцатеричные, заменяя, например, 2581 на 0xb23. Или предположим, в нашем прежнем примере мы не уверены в наличии названий для всех версий, поэтому все остальные хотим оставить без изменений. Творчески применив форматирование /x, можно сказать:

```
s{
    version
    \s+
    (
        [0-9.]+
    )
}{
    $Names{$1}
        ? "the $Names{$1} release"
        : $&
}xge;
```

Правая часть s///e (или, в данном варианте, нижняя часть) проходит синтаксическую проверку и компилируется на этапе компиляции программы в целом. Синтаксические ошибки обнаруживаются во время компиляции, а исключительные ситуации времени выполнения не перехватываются. Каждый дополнительный модификатор /e после первого (например, /ee, /eee и так далее) эквивалентен вызову eval *STRING* над результатом выполнения кода, по одному вызову на каждый дополнительный модификатор /e. При этом вычисляется результат выражения, представленного кодом, а информация об исключительных ситуациях сохраняется в специальной переменной $@. Дополнительные сведения можно найти в разделе «Программные шаблоны» далее в этой главе.

Модификация строк en passant[1]

Иногда требуется, чтобы новая, модифицированная строка не портила старую, на которой она основана. Вместо того чтобы писать:

```
$lotr = $hobbit;
$lotr =~ s/Bilbo/Frodo/g;
```

можно объединить эти команды в одну. Порядок старшинства операторов требует, чтобы присваивание было заключено в круглые скобки, как в большинстве случаев нетривиального применения =~ к выражению.

```
($lotr = $hobbit) =~ s/Bilbo/Frodo/g;
```

[1] На проходе (фр.) – *Прим. перев.*

В отсутствие скобок была бы изменена строка $hobbit, а в $lotr мы получили бы количество произведенных замен, так что продолжение книги «Хоббит» вышло бы довольно скучное.

И, да, в настоящее время можно с тем же успехом использовать модификатор /r:

```
$lotr = $hobbit =~ s/Bilbo/Frodo/gr;
```

Но многие поклонники Perl по-прежнему используют старую идиому.

Модификация массивов en masse[1]

Применить оператор s/// непосредственно к массиву нельзя. Для этого нужен цикл. По счастливому совпадению, свойство циклом for/foreach создавать псевдонимы в сочетании с использованием $_ в качестве переменной цикла по умолчанию лежит в основе стандартной идиомы Perl для поиска и замены в каждом элементе массива:

```
for (@chapters) { s/Bilbo/Frodo/g } # Сделать замену в каждой главе
s/Bilbo/Frodo/g for @chapters;      # То же самое.
```

Как и для простых скалярных переменных, можно сочетать подстановку с присваиванием, если требуется сохранить исходные значения:

```
@oldhues = ('bluebird', 'bluegrass', 'bluefish', 'the blues');
for (@newhues = @oldhues) { s/blue/red/ }
print "@newhues\n";        # выведет redbird redgrass redfish the reds
```

Другой способ проделать то же самое: объединить модификатор /r (он появился в v5.14) с оператором map:

```
@newhues = map { s/blue/red/r } @oldhues;
```

Идиоматический способ выполнения нескольких подстановок для одной переменной является применение цикла из одной итерации. Вот пример приведения в каноническую форму пробельных символов в строке:

```
for ($string) {
    s/^\s+//;          # отбросить пробельные символы в начале строки
    s/\s+$//;          # отбросить замыкающие пробельные символы
    s/\s+/ /g;         # сжать внедренные пробельные последовательности
}
```

что, по стечению обстоятельств, дает такой же результат, как и

```
$string = join(" ", split " ", $string);
```

Или можно организовать такой же цикл с присваиванием, какой мы применили для массива:

```
for ($newshow = $oldshow) {
    s/Fred/Homer/g;
    s/Wilma/Marge/g;
    s/Pebbles/Lisa/g;
    s/Dino/Bart/g;
}
```

[1] Массовая (фр.) – *Прим. перев.*

Когда глобальная подстановка оказывается недостаточно глобальной

Иногда модификатора /g оказывается недостаточно, чтобы произвести все необходимые изменения: то исходная строка содержит перекрывающиеся соответствия, то подстановку хочется производить справа налево, то нужно, чтобы длина $` изменялась от одного соответствия к другому. Как правило, повторные вызовы s/// позволяют добиться нужного результата. Однако хорошо бы, чтобы цикл прекратился, когда s/// не найдет решительно ничего, поэтому подобную конструкцию мы заключаем в условный оператор, а в теле цикла, выходит, делать уже нечего. Поэтому мы просто пишем в теле цикла 1, что довольно скучно, но иногда скука – это лучшее, на что можно рассчитывать. Вот несколько примеров использования этих странных зверюшек из регулярных выражений, продолжающих возникать тут и там:

```
# расставить в нужных местах целого числа запятые
1 while s/(\d)(\d\d\d)(?!\d)/$1,$2/;

# заменить табуляцию на пробелы по 8 в колонке
1 while s/\t+/" " x (length($&)*8 - length($`)%8)/e;

# удалить (вложенные (даже глубоко вложенные (как это))) замечания
1 while s/\([^()]*\)//g;

# удалить слова, повторяющиеся дважды (и трижды (и четырежды...))
1 while s/\b(\w+) \1\b/$1/gi;
```

В последнем случае требуется цикл, иначе

```
Paris in THE THE THE THE spring.
```

превратится в

```
Paris in THE THE spring.
```

что может вызвать перед глазами того, кто немного знает французский, образ Париса, сидящего в артезианском колодце с ледяным чаем, ведь «the» по-французски – «чай».[1] Парижанина, конечно, не обманешь.

Оператор транслитерации tr///

```
tr/SEARCHLIST/REPLACEMENTLIST/cdsr

 LVALUE =~ tr/SEARCHLIST/REPLACEMENTLIST/cds
 RVALUE =~ tr/SEARCHLIST/REPLACEMENTLIST/cdsr
 RVALUE =~ tr/SEARCHLIST//c
```

Поклонникам *sed* посвящается оператор y///, в качестве синонима tr///. По этой причине нельзя вызывать функции с именем y, как и функции с именами q или m. Во всех остальных отношениях y/// совпадает с tr///, и больше мы об этом говорить не будем.

Может показаться, что этот оператор не вписывается в главу о поиске по шаблону, поскольку в нем не используются шаблоны. Он просматривает строку символ за символом и заменяет каждый символ, оказывающийся в *SEARCHLIST* (это не регу-

[1] Spring – весна; родник; Paris – Париж; Парис, сын Приама. – *Прим. перев.*

лярное выражение), на соответствующий символ из *REPLACEMENTLIST* (это не строка подстановки). Однако этот оператор несколько похож на m// и s///, и с ним даже можно применять операторы привязки =~ или !~, почему мы и описываем его здесь. (Операторы qr// и split выполняют сопоставление с шаблоном, но с ними не используются операторы привязки, поэтому они находятся в другом месте книги. Вот так все непросто.)

Транслитерация возвращает число замененных или удаленных символов. Если строка не задана явным образом, с помощью операторов =~ или !~, изменения вносятся в строку $_. В *SEARCHLIST* и *REPLACEMENTLIST* можно определять диапазоны последовательных символов с помощью дефиса:

```
$message =~ tr/A-Za-z/N-ZA-Mn-za-m/;   # шифрование rot13.
```

Обратите внимание, что диапазон вроде A-Z предполагает линейный набор символов типа ASCII. Но в каждом наборе символов есть свои представления о том, как они упорядочены, и, следовательно, о том, какие символы попадают в конкретный диапазон. Разумный принцип состоит в использовании только таких диапазонов, начало и конец которых находятся в том же алфавите либо в одном регистре (a–e, A–E), или являются цифрами (0–4). Ко всем остальным следует относиться с подозрением. Если есть сомнения, укажите набор символов явным образом: ABCDE. Даже применение простых диапазонов, таких как [ᴀ-ᴇ], может закончиться провалом, тогда как применение полных наборов символов вида [ᴀʙᴄᴅᴇ] будет давать положительные результаты, потому что коды малых заглавных латинских букв не следуют по порядку, как показано в табл. 5.4

Таблица 5.4. Коды малых заглавных латинских букв

Глиф	Код	Категория	Алфавит	Название
ᴀ	U+1D00	GC=Ll	SC=Latin	LATIN LETTER SMALL CAPITAL A
ʙ	U+0299	GC=Ll	SC=Latin	LATIN LETTER SMALL CAPITAL B
ᴄ	U+1D04	GC=Ll	SC=Latin	LATIN LETTER SMALL CAPITAL C
ᴅ	U+1D05	GC=Ll	SC=Latin	LATIN LETTER SMALL CAPITAL D
ᴇ	U+1D07	GC=Ll	SC=Latin	LATIN LETTER SMALL CAPITAL E

В *SEARCHLIST* и *REPLACEMENTLIST* интерполяция переменных не производится, как в строках в двойных кавычках; можно, однако, использовать последовательности с обратной косой чертой, соответствующие конкретным символам, например \n или \015.

В табл. 5.5 перечислены модификаторы, применимые к оператору tr///. Они совершенно отличны от тех, которые применяются к m//, s/// или qr//, даже если выглядят так же.

Таблица 5.5. Модификаторы tr///

Модификатор	Значение
/c	Дополнение *SEARCHLIST*
/d	Удалить найденные, но не замененные символы
/s	Сжать повторяющиеся замененные символы
/r	Вернуть результат транслитерации, не изменять исходную строку

Если задан модификатор /r, транслитерации подвергается новая копия строки, которая и возвращается. Она может не быть l-значением.

```
say "Drogo" =~ tr/Dg/Fd/r;      # Drogo -> Frodo
```

Если задан модификатор /c, осуществляется дополнение набора символов в *SEARCHLIST*; т.е. фактический список для поиска состоит из всех символов, *отсутствующих* в *SEARCHLIST*. В случае применения Юникода таких символов может оказаться *очень много*, но, поскольку они хранятся логически, а не физически, можно не беспокоиться о том, что не хватит памяти.

Модификатор /d превращает tr/// в оператор «замены-удаления»: любые символы, указанные в *SEARCHLIST*, для которых не задана замена в *REPLACEMENTLIST*, удаляются. (Это дает несколько большую гибкость, чем ключ -d в некоторых программах *tr*(1), удаляющих только то, что они находят в *SEARCHLIST*.)

Если задан модификатор /s, то последовательности символа, превращаемого в тот же символ, сжимаются до единственного символа.

При использовании модификатора /d список *REPLACEMENTLIST* всегда интерпретируется точно так, как он задан. В отсутствие этого модификатора, если *REPLACEMENTLIST* короче *SEARCHLIST*, последний символ *REPLACEMENTLIST* размножается до достижения нужной длины. Если *REPLACEMENTLIST* представляет собой пустую строку, в него копируется *SEARCHLIST*, что полезно, если требуется просто подсчитать символы, не заменяя их. Это полезно также для сжатия символов посредством /s. Если требуется всего лишь подсчитать количество символов, можно просто использовать *RVALUE* вместо *LVALUE*.

```
tr/aeiou/!/;                  # заменить все гласные на!
tr{/\\\r\n\b\f. }{_};         # заменить необычные символы на подчеркивание

$count = ($para =~ tr/\n//);  # подсчитать количество переводов строки в $para
$count = tr/0-9//;            # подсчитать цифры в $_

tr/@$%*//d;                   # удалить указанные

# изменить en passant
($HOST = $host) =~ tr/a-z/A-Z/;

# результат тот же, но как r-значение
$HOST = ($host =~ tr/a-z/A-Z/r);

$pathname =~ tr/a-zA-Z/_/cs;  # заменить небуквы (ASCII) на подчеркивание
```

Если в *SEARCHLIST* один и тот же символ повторяется несколько раз, то используется только первый, поэтому команда

```
tr/AAA/XYZ/
```

поменяет все символы A на X (в $_).

Хотя переменные не интерполируются в tr///, того же эффекта можно добиться с помощью eval *EXPR*:

```
$count = eval "tr/$oldlist/$newlist/";
die if $@;    # распространяет исключение при недопустимом содержимом eval
```

Еще одно замечание: не используйте tr/// для перевода текста в верхний или нижний регистр. В этом случае следует предпочесть последовательности \U или \L в строках, заключенных в двойные кавычки (или эквивалентные функции uc и lc), поскольку они учитывают национальные установки и Юникод, а tr/a-z/A-Z/ не учитывает. Кроме того, в строках Юникода последовательность \u и соответствующая ей функция ucfirst учитывают понятие «заглавного регистра» (titlecase), который для некоторых символов отличается от верхнего регистра.

Последовательность \F соответствует функции fc; см. описание fc в главе 27. Они появились в Perl v5.16 и служат для сравнения без учета регистра символов, например: "\F$a" eq "\F$b", или эквивалентное выражение, fc($a) eq fc($b). Чтобы обеспечить сопоставление без учета регистра, всегда применялся модификатор /i, внутренне использующий механизм свертки; теперь появились последовательность \F и функция fc, обеспечивающие более прямолинейный доступ к этому механизму. См. также раздел «Сравнение и сортировка текста Юникода» в главе 6.

Метасимволы и метазнаки

Теперь, полюбовавшись на все замысловатые клетки, можем продолжить разглядывать тварей, которые в этих клетках находятся – на те забавные значки, которые помещают внутрь шаблонов. Думается, сейчас вы уже свыклись с тем обстоятельством, что эти значки не являются обычным программным кодом на Perl, таким как вызовы функций или арифметические операторы. Регулярные выражения представляют собой маленький самостоятельный язык, укрытый в недрах Perl. (В каждом из нас есть немного джунглей.)

При всей своей мощи и выразительности шаблоны в Perl распознают те же 12 обычных метасимволов, которые имеются во многих других пакетах, поддерживающих регулярные выражения:

```
\ | ( ) [ { ^ $ * + ?
```

Некоторые из них меняют правила, делая особенными следующие за ними символы, которые при других обстоятельствах ничем особенным не отличаются. Мы не хотим называть длинные последовательности символами, поэтому, когда такие последовательности образуются, мы называем их *метазнаками* (*metasymbols*, или иногда просто *symbols*). Но на верхнем уровне эти двенадцать метасимволов представляют собой все, о чем вы (и Perl) должны думать. Все остальное проистекает из них.

Некоторые простые метасимволы (., ^, $) действуют самостоятельно. Они не оказывают непосредственного влияния на соседние символы. Другие метасимволы, например \, действуют как префиксные операторы, управляя тем, что следует за ними. Третьи, например, *, + и ?, действуют как постфиксные операторы, управляя тем, что непосредственно предшествует им. Один метасимвол, |, действует как инфиксный оператор, располагаясь между подвластными ему операндами. Есть даже структурирующие метасимволы, действующие как охватывающие операторы и управляющие тем, что в них содержится; например, к ним относятся (...) и [...]. Особенно важны круглые скобки, поскольку они ограничивают | изнутри, а *, + и ? снаружи.

Если запоминать только один из этих двенадцати метасимволов, следует запомнить обратную косую черту. (Ну... еще круглые скобки.) Это связано с тем, что обратная косая черта отключает все остальные. Когда она предшествует в шаблоне символу, не являющемуся буквой или цифрой, то превращает этот символ в литерал. Если необходимо найти по шаблону один из этих двенадцати метасимволов буквально, его нужно предварить обратной косой чертой. Поэтому \. соответствует действительной точке, \$ – действительному знаку доллара, \\ – действительной обратной косой черте и т. д. Это называют «экранированием» метасимвола («escaping», «quoting» или «backslashing»). (Конечно, вы уже знаете, что обратная косая черта используется для подавления интерполяции переменных в строках, заключенных в двойные кавычки.)

Обратная косая черта превращает метасимвол в литеральный, но действие его на последующий буквенно-цифровой символ – иное. Что было без обратной косой черты обычным, превращается в особенное. Это значит, что вместе они образуют метазнак. Алфавитный список этих метазнаков можно найти ниже, в табл. 5.9.

Таблицы метазнаков

В последующих таблицах в колонке «Атомарный» указывается «да», если метазнак поддается количественному определению (может соответствовать чему-то, обладающему в какой-то мере шириной). Мы также применяли многоточие (...) для представления «чего-то еще». (Пожалуйста, прочтите приводимое далее описание, чтобы понять, что означает , если это было непонятно из однострочного пояснения в таблице.)

В табл. 5.6 приведены основные обычные метазнаки. Первые четыре из них являются структурирующими, и о них говорилось выше, тогда как последние три – это просто метасимволы. Примером атома служит метасимвол . (точка), поскольку он соответствует чему-то, имеющему ширину (в данном случае ширину символа); ^ и $ представляют собой примеры утверждений, поскольку соответствуют чему-то, имеющему нулевую ширину, и вычисляются, только чтобы выяснить, истинны они или нет.

Таблица 5.6. Общие метасимволы регулярных выражений

Символ	Атомарный	Значение
\...	По-разному	Сделать обычным следующий не буквенно-цифровой метасимвол и (возможно) сделать метасимволом следующий буквенно-цифровой символ
...|...	Нет	Перечисление (соответствие одному или другому)
(...)	Да	Группа (рассматривается как единое целое)
[...]	Да	Класс символов (соответствие одному символу из набора)
^	Нет	Истинно в начале строки (или, возможно, после любого символа перевода строки)
	Да	Соответствие одному символу (обычно, кроме символа перевода строки)
$	Нет	Истинно в конце строки (или, возможно, перед любым символом перевода строки)

Квантификаторы, которым ниже посвящен отдельный раздел, показывают, сколько раз предшествующий атом (т.е. отдельный символ или группа символов) должен быть найден. Они перечислены в табл. 5.7.

Таблица 5.7. Квантификаторы регулярных выражений

Максимальный	Минимальный	Неуступающий	Допустимый диапазон
{*MIN*,*MAX*}	{*MIN*,*MAX*}?	{*MIN*,*MAX*}?+	Не меньше *MIN* соответствий, но не больше *MAX*
{*MIN*,}	{*MIN*,}?	{*MIN*,}?+	Не меньше *MIN* соответствий
{*COUNT*}	{*COUNT*}?	{*COUNT*}?+	Ровно *COUNT* соответствий
*	*?	*+	0 или более соответствий (то же, что и {0,})
+	+?	++	1 или более соответствий (то же, что и {1,})
?	??	?+	0 или 1 соответствий (то же, что и {0,1})

Минимальный квантификатор пытается найти как можно *меньше* символов в отпущенных ему пределах, максимальный – как можно *больше*.

Например, .+ обеспечивает соответствие по крайней мере одному символу в строке, но будет соответствовать всем им, если дать возможность. Эти возможности описываются далее в разделе «Маленький Механизм, который /(не)? может/» данной главы.

Неуступающий квантификатор действует как максимальный, за исключением того, что не уступает захваченные символы механизму возвратов, тогда как минимальный и максимальный квантификаторы могут уступать.

Заметьте, что квантификаторы нельзя квантифицировать. Обозначения, подобные ?? и ++, являются самостоятельными квантификаторами, минимальным и неуступающим, соответственно, а не односимвольным квантификатором, квантифицированным другим квантификатором. Квантификаторы могут определять только количество атомов, но сами квантификаторами атомами не являются.

Мы стремились обеспечить расширяемый синтаксис для новых типов метазнаков. Учитывая, что в нашем распоряжении была всего лишь дюжина метасимволов, мы выбрали для конструирования произвольных синтаксических расширений прежде недопустимую для регулярных выражений последовательность. Все эти метазнаки, за исключением последнего, имеют вид (?*KEY*...); т.е. (сбалансированная, имеющая пару) скобка, за которой следует вопросительный знак, за которым следует *KEY* и оставшаяся часть подшаблона. Символ *KEY* указывает на конкретное расширение синтаксиса регулярных выражений. Полный их список приведен в табл. 5.8. В основном здесь структурирующие символы, поскольку они построены на скобках, но у них есть и дополнительные значения. И снова, квантифицироваться могут только атомы, поскольку они представляют нечто действительно существующее (потенциально).

Таблица 5.8. Расширенные последовательности регулярных выражений

<table>
<tr><th>Расширение</th><th>Атомарный</th><th>Значение</th></tr>
<tr><td>(?#...)</td><td>Нет</td><td>Комментарий, отбрасывается</td></tr>
<tr><td>(?:...)</td><td>Да</td><td>Несохраняющая группа</td></tr>
<tr><td>(?>...)</td><td>Да</td><td>Группировка, не сохраняющая и не уступающая символы из найденного соответствия</td></tr>
<tr><td>(?adlupimsx-imsx)</td><td>Нет</td><td>Включить/отключить модификаторы шаблона</td></tr>
<tr><td>(?^alupimsx)</td><td>Нет</td><td>Сбрасывает и включает модификаторы шаблона</td></tr>
<tr><td>(?adlupimsx-imsx:...)</td><td>Да</td><td>Несохраняющая группа, плюс включение/выключение модификаторов шаблона</td></tr>
<tr><td>(?^alupimsx:...)</td><td>Да</td><td>Несохраняющая группа, плюс сброс и включение модификаторов шаблона</td></tr>
<tr><td>(?=...)</td><td>Нет</td><td>Истина, в случае успеха опережающей проверки</td></tr>
<tr><td>(?!...)</td><td>Нет</td><td>Истина, в случае неудачи опережающей проверки</td></tr>
<tr><td>(?<=...)</td><td>Нет</td><td>Истина, в случае успеха ретроспективной проверки</td></tr>
<tr><td>(?<!...)</td><td>Нет</td><td>Истина, в случае неудачи ретроспективной проверки</td></tr>
<tr><td>(?|...|...|...)</td><td>Да</td><td>Выбор ветви для нумерованных групп</td></tr>
<tr><td>(?<<i>NAME</i>>...)</td><td>Да</td><td>Именованная сохраняющая группировка; также можно использовать форму записи (?'<i>NAME</i>'...). См. \k<<i>NAME</i>> ниже</td></tr>
<tr><td>(?{...})</td><td>Нет</td><td>Выполнить внедренный программный код на языке Perl</td></tr>
<tr><td>(??{...})</td><td>Да</td><td>Поиск соответствия регулярному выражению из внедренного кода на Perl</td></tr>
<tr><td>(?NUMBER)</td><td>Да</td><td>Вызывает независимое подвыражение в группе NUMBER; также можно использовать форму записи (?+NUMBER), (?-NUMBER), (?0) и (?R). Использовать амперсанд здесь нельзя</td></tr>
<tr><td>(?&NAME)</td><td>Да</td><td>Рекурсия в группе NAME; здесь должен использоваться амперсанд; также можно использовать форму записи (?P>NAME)</td></tr>
<tr><td>(?(COND)...|...)</td><td>Да</td><td>Поиск по шаблону if-then-else</td></tr>
<tr><td>(?(COND)...)</td><td>Да</td><td>Поиск по шаблону if-then</td></tr>
<tr><td>(?(DEFINE)...)</td><td>Нет</td><td>Определение именованной группы для последующего обращения как к «подпрограмме» (?&NAME)</td></tr>
<tr><td>(*VERB)</td><td>Нет</td><td>Глагол управления механизмом возвратов; также можно использовать форму записи (*VERB:NAME)</td></tr>
</table>

Команды управления механизмом возвратов все еще находятся на стадии экспериментальных и потому не будут обсуждаться здесь. Тем не менее, вы можете столкнуться с ними рано или поздно, сунув свой нос в дела волшебников. Поэтому обязательно ознакомьтесь со страницей *perlre* справочного руководства, если встретите любую из следующих команд:

```
(*ACCEPT)
(*COMMIT)
(*FAIL)      (*F)
```

```
(*MARK:NAME) (*:NAME)
(*PRUNE)     (*PRUNE:NAME)
(*SKIP)      (*SKIP:NAME)
(*THEN)      (*THEN:NAME)
```

Или просто бегите прочь.

И наконец, в табл. 5.9 перечислены все ваши любимые буквенно-цифровые метазнаки. (Знаки, обрабатываемые на этапе интерполяции переменных, помечены в колонке «атомарный» символом тире «–», поскольку Механизм их даже не увидит.)

Таблица 5.9. Буквенно-цифровые метазнаки регулярных выражений

Символ	Атомарный	Значение
\0	Да	Соответствует с нулевому символу (U+0000, NULL, NUL)
\NNN	Да	Соответствует символу, заданному в восьмеричной системе, до \377
\n	Да	Соответствует n-й сохраняющей группе (десятичное)
\a	Да	Соответствует символу тревоги (ALERT, BEL)
\A	Нет	Истина, если в начале строки
\b	Да	Соответствует символу забоя (BACKSPACE, BS) (только в символьных классах)
\b	Нет	Истина, если граница слова
\B	Нет	Истина, если не граница слова
\cX	Да	Соответствует управляющему символу Control-X (\cZ, \c[и т. д.)
\C	Да	Соответствует одному байту (C char) даже в UTF-8 (опасно!)
\d	Да	Соответствует любой цифре
\D	Да	Соответствует любому нецифровому символу
\e	Да	Соответствует символу ESCAPE, ESC.
\E	–	Конец трансляции регистра (\F, \L, \U) или метакавычки (\Q)
\f	Да	Соответствует символу новой страницы (FORM FEED, FF)
\F	–	Приведение к единому регистру всех символов до метасимвола \E[a]
\g{GROUP}	Да	Соответствует именованной или нумерованной сохраняющей группе
\G	Нет	Истинно в точке окончания предыдущего соответствия при поиске m//g
\h	Да	Соответствует любому горизонтальному пробельному символу
\H	Да	Соответствует любому символу, кроме горизонтального пробельного символа
\k{GROUP}	Да	Соответствует именованной сохраняющей группе, также можно использовать форму записи \k'NAME'
\K	Нет	Исключить из совпадения текст слева от \K
\l	–	Перевести следующий символ в нижний регистр (не свертка)

[a] Метасимвол \F и соответствующая ему функция fc появились в Perl v5.16.

Таблица 5.9 (продолжение)

Символ	Атомарный	Значение
\L	–	Перевести в нижний регистр (не свертка) текст до \E
\n	Да	Соответствует символу новой строки (обычно LF, LINE FEED)
\N	Да	Соответствует любому символу, кроме символа новой строки
\N{*NAME*}	Да	Соответствует именованному символу, псевдониму или последовательности, например: \N{greek:Sigma} для «Σ»
\o{*NNNN*}	Да	Соответствует символу с указанным восьмеричным кодом
\p{*PROP*}	Да	Соответствует любому символу с указанным свойством
\P{*PROP*}	Да	Соответствует любому символу, не имеющему указанного свойства
\Q	–	Добавить обратную косую черту перед всеми последующими символами, не являющимися буквами или цифрами, вплоть до \E
\r	Да	Соответствует символу возврата каретки (обычно CARRIAGE RETURN, CR)
\R	Да	Соответствует любой графеме, обозначающей разрыв строки (только не в символьных классах)
\s	Да	Соответствует любому пробельному символу
\S	Да	Соответствует любому непробельному символу
\t	Да	Соответствует символу табуляции (CHARACTER TABULATION, HT)
\u	–	Перевести следующий символ в заглавный (не верхний) регистр
\U	–	Перевести в верхний (не заглавный) регистр текст до \E
\v	Да	Соответствует любому вертикальному пробельному символу
\V	Да	Соответствует любому символу, не являющемуся вертикальным пробельным символом
\w	Да	Соответствует любым «символам слова» (буквенно-цифровым, комбинационным знакам и соединителям)
\W	Да	Соответствует любому символу, не являющемуся «символом слова»
\x{*abcd*}	Да	Соответствует символу с указанным шестнадцатеричным кодом
\X	Да	Соответствует графеме (только не в символьных классах)
\z	Нет	Истина, если конец строки
\Z	Нет	Истина, если конец строки или перед символом перевода строки

Если в \p или \P имя свойства состоит из одного символа, фигурные скобки не обязательны. Если в \x шестнадцатеричное число состоит из двух и менее цифр, фигурные скобки не обязательны. Отсутствие фигурных скобок в \N означает соответствие любому символу, кроме символа перевода строки. В \g можно не использовать скобки, если вы ссылаетесь на нумерованную группу (однако мы рекомендуем использовать их всегда).

Метасимвол \R соответствует либо символу CARRIAGE RETURN, за которым следует символ LINE FEED (без уступки механизму возвратов), либо любому другому вертикальному пробельному символу. Эквивалентен выражению (?>\r\n|\v). Отсутствие уступки механизму возвратов означает, что "\r\n" =~ /\R\n/ может завершиться безрезультатно; обнаружив один раз пару символов CRLF, Механизм позднее не изменит свое поведение для поиска самостоятельного символа CARRIAGE RETURN,

даже если далее в шаблоне будет находиться фрагмент, требующий присутствия символа LINE FEED для соответствия всего шаблона.

В классах символов (квадратных скобках) могут использоваться только метасимволы, в описании которых присутствует «Соответствует символу...» или «Соответствует любому ...», и только если они соответствуют одному символу, поэтому метасимволы \R и \X нельзя использовать в символьных классах. Это значит, что класс символов может соответствовать только одному символу в каждый момент и содержать только метазнаки, описывающие конкретные отдельные символы. Конечно, эти метасимволы можно использовать вне классов символов наряду с остальными неклассифицирующими метазнаками. Обратите внимание, что \b – это два совершенно разных «зверя»: внутри класса символов это символ забоя (backspace), а вне его – утверждение границы слова.

Метасимвол \K (от: «Keep» – сохранить то, что уже совпало) не соответствует чему-либо. Он просто сообщает механизму регулярных выражений оставить часть соответствия и действует подобно переменной $& или ${^MATCH}, или левой стороне операции подстановки. См. примеры в разделе с описанием оператора s///.

Существует некое перекрытие между символами, которые может находить шаблон, и символами, которые может интерполировать обычная строка в двойных кавычках. Поскольку регулярные выражения обрабатываются за два прохода, иногда возникает неоднозначность в том, на котором из этапов должен обрабатываться данный символ. Если возникает неоднозначность, то на этапе интерполяции переменных интерпретация таких символов откладывается для анализатора регулярных выражений.

Однако на этапе интерполяции переменных можно отложить интерпретацию до анализатора регулярных выражений, только если известно, что проводится анализ регулярного выражения. Можно задавать регулярные выражения как обычные строки в двойных кавычках, но тогда нужно следовать правилам, применяемым к обычным строкам в двойных кавычках. Все предыдущие метазнаки, соответствующие фактическим символам, будут работать, даже если они не откладываются до анализатора регулярных выражений. Но нельзя использовать другие метасимволы в обычных двойных кавычках (или аналогичных конструкциях типа `...`, qq(...), qx(...) или эквивалентных внедренных документах (here documents)). Чтобы строка анализировалась как регулярное выражение без поиска соответствий, следует использовать оператор qr// (quote regex, цитировать или заключить в кавычки регулярное выражение).

С другой стороны, escape-последовательности трансляции регистра и метакавычек (\U и компания) *должны* обрабатываться на этапе интерполяции переменных, поскольку назначение этих метазнаков состоит в воздействии на способ интерполяции переменных. Если подавить интерполяцию переменных с помощью одинарных кавычек, не будет возможность применять и escape-последовательности трансляции. Ни переменные, ни escape-последовательности трансляции (\U и прочие) не разыменовываются ни в строках в одинарных кавычках, ни в операторах одинарных кавычек m'...' или qr'...'. Даже при выполнении интерполяции эти escape-последовательности трансляции игнорируются, если они фигурируют в качестве результата интерполяции переменных, поскольку в этом случае уже слишком поздно воздействовать на интерполяцию переменных.

Хотя оператор транслитерации не принимает регулярные выражения, любой метазнак из обсуждавшихся нами, соответствующий конкретному отдельному

символу, действует и в операции tr///. Остальные – нет (за исключением обратной косой черты, которая, как обычно, продолжает работать «от обратного»).

Конкретные символы

Как уже говорилось выше, любой фрагмент шаблона, не обладающий специальным назначением, соответствует самому себе. Это значит, что /a/ соответствует «a», /=/ соответствует «=» и т. д. Однако некоторые символы не так просто ввести с клавиатуры, а если вам это и удастся, они только испортят вывод текста на вашем экране (Это если повезет. Управляющие символы славятся своим буйным нравом.) Чтобы справиться с этим обстоятельством, в регулярном выражении Perl распознает псевдонимы символов, перечисленные в табл. 5.10.

Таблица 5.10. Псевдонимы управляющих символов, распознаваемые в двойных кавычках

Псевдоним	Значение
\0	Нулевой символ (NUL, NULL)
\a	Тревога (BEL, ALERT)
\e	Символ экранирования (ESCAPE, ESC)
\f	Символ новой страницы (FORM FEED, FF)
\n	Символу перевода строки (LF, LINE FEED)
\r	Символ возврата каретки (CARRIAGE RETURN, CR)
\t	Символ табуляции (CHARACTER TABULATION, HT)

Как и в строках, заключенных в двойные кавычки, Perl дополнительно отличает в шаблонах следующие пять метазнаков:

\c*X*

Именованный управляющий символ ASCII, например, \cC для Control-C, \cZ для Control-Z, \c[для ESC и \c? для DEL. Допускаются целые числа в диапазоне 0–31, а также 127.

NNN

Символ, заданный с помощью двух- или трехзначного восьмеричного кода. Начальный 0 обязателен только для чисел, меньших 010 (8 десятичное), поскольку (в отличие от случая строк в двойных кавычках) код из одной цифры всегда считается ссылкой на подстроку текста, сохраненную группой в шаблоне. Несколько цифр интерпретируются как *n*-я ссылка, если ранее в шаблоне вы сохранили не менее *n* подстрок (где *n* считается десятичным числом); в противном случае они интерпретируются как символ в восьмеричной системе.

\x{*HEXDIGITS*}

Кодовый пункт (номер) символа, заданный в виде одной или двух шестнадцатеричных цифр ([0-9a-fA-F]), например, \x1B. Формат из одной цифры можно использовать, только если следующий символ не является шестнадцатеричной цифрой. Если применяются фигурные скобки, цифр может быть сколько угодно. Например, \x{262f} соответствует символу Юникода «Инь – Ян» U+262F YIN YANG (☯).

\N{*NAME*}

Именованный символ, псевдоним, или последовательность; например, \N{GREEK SMALL LETTER EPSILON}, \N{greek:epsilon} или \N{epsilon}. Чтобы такие последовательности распознавались, должна действовать прагма charnames, описанная в главе 29. Эта прагма определяет также разновидности имен, которые можно использовать (":full" соответствует первой форме записи выше, а ":short" – двум другим).

Символы можно также определять с помощью нотации \N{U+*NUMBER*}. Например, \N{U+263B} соответствует символу ☻, BLACK SMILING FACE. Такое применение не требует прагмы charnames.

Список всех имен символов Юникода можно найти в ближайшем документе с описанием стандарта Юникода или сгенерировать перебором charnames::viacode(*N*), для *N* в диапазоне от 0 до 0x10_FFFF, не забывая пропускать суррогатные символы.

\o{*NUMBER*}

Символ, определяемый восьмеричным кодом. В отличие от неоднозначной формы записи *NNN*, в этой нотации можно указывать любое количество восьмеричных цифр, которые никогда не будут перепутаны со ссылкой на сохраняющую группу.

Метазнаки-маски

Три особых метазнака служат масками, каждой из которых соответствует «любой» символ (для заранее определенных значений слова «любой»). Это метазнаки точки ("."), \C и \X. Ни один из них нельзя использовать в символьных классах. В символьных классах не допускается наличие точки, потому что она будет соответствовать (почти) любому существующему символу, являясь своего рода всеобщим классом символов. Если необходимо все исключить или все включить, нет особого смысла в использовании класса символов. Особые маски \C и \X имеют особый структурирующий смысл, который не очень хорошо согласуется с идеей выбора одного символа Юникода, а именно на таком уровне работают классы символов.

Метасимвол «точка» соответствует любому отдельному символу, кроме символа перевода строки. (А с модификатором /s соответствует и ему тоже. В этом случае используйте метасимвол \N для поиска соответствия любому символу, кроме перевода строки.) Как и для любого из двенадцати спецсимволов в шаблоне, для буквального поиска точки ее нужно экранировать с помощью обратной косой черты. Например, следующая команда проверяет, заканчивается ли имя файла точкой и расширением из одного символа:

```
if ($pathname =~ /\.(.)\z/s) {
    say "Оканчивается на $1";
}
```

Первая точка, экранированная обратной косой чертой, представляет собой литеральный символ, а вторая описывает «соответствие любому символу». Последовательность \z указывает, что соответствие следует искать только в конце строки, а модификатор /s разрешает точке соответствовать и символу перевода строки. (Да, использование символа перевода строки в качестве расширения файла – это Не Очень Вежливо, но может случиться и такое.)

Метасимвол точки чаще всего используется с квантификатором. Выражение .* соответствует максимальному числу символов, тогда как .*? соответствует минимальному числу символов. Но иногда она применяется и без квантификатора, ради ее ширины: выражение /(..):(..):(..)/ ищет три поля, разделенных двоеточием, каждое из которых имеет размер в два символа.

Не путайте символы с байтами. В былые времена точка соответствовала единственному байту, но сейчас она соответствует символам Юникода, многие из которых невозможно закодировать одним байтом:

```
use charnames qw[ :full ];
$BWV[887] = "G\N{MUSIC SHARP SIGN} minor";
my ($note, $black, $mode) = $BWV[887] =~ /^([A-G])(.)\s+(\S+)/;
say "Так выглядит знак диез!" if $black eq chr 0x266f; # ♯
```

Метазнак \X соответствует символу в более широком смысле. Он соответствует строке из одного или нескольких символов Юникода, называемую «grapheme cluster» – кластер графем. Его назначение – захват последовательности символов, визуально образующих единый глиф. Обычно она состоит из базового символа, за которым следуют комбинационные и диакритические знаки, такие как седиль «¸» и умляут «¨», объединяющиеся с этим базовым символом в одну логическую единицу. Также это может быть последовательность Юникода, обозначающая разрыв строки, включая "\r\n", или, ввиду того, что диакритики не сочетаются с концами строк, самостоятельный комбинационный или диакритический знак в начале строки.

Изначальный метазнак \X почти всегда был эквивалентен выражению (?>\PM\pM*), но это не вполне качественное определение, поэтому стандарту Юникод пришлось уточнить понятия кластеров графем. Точное определение достаточно сложное, но оно близко к следующему:

```
(?> \R
    | \p{Grapheme_Base} \p{Grapheme_Extend}*
    | \p{Grapheme_Extend}+ |
)
```

Суть в том, что \X ищет один видимый символ (графему), даже если для программиста этот символ состоит из нескольких отдельных символов (кодовых пунктов). Длина соответствия /\X/ может быть больше одного символа, если \R в псевдоопределении выше совпадет с парой CRLF или если базовый символ графемы сопровождается одним или более дополнительными символами.[1] Неуступающая группировка означает, что \X уже не передумает, обнаружив базовый символ, за которым следуют любые расширяющие символы. Например, /\X.\z/ никогда не найдет "cafe\x{301}", где U+0301 – это COMBINING ACUTE ACCENT, потому что \X не уступает символы механизму возвратов.

Если вы работаете с Юникодом и действительно хотите получить один байт, а не один символ, *можете* воспользоваться метазнаком \C. Он всегда соответствует одному байту (именно одному значению типа char в языке C), даже если при этом возникает расхождение с потоком символов Юникода. См. соответствующие предупреждения в главе 6. Однако навряд ли это правильный путь решения задачи.

---

[1] Обычно комбинационные знаки; в настоящее время единственными незнаковыми графемами, расширяющими символы, являются ZERO WIDTH NONJOINER, ZERO WIDTH JOINER, HALFWIDTH KATAKANA VOICED SOUND MARK и HALFWIDTH KATAKANA SEMIOICED SOUND MARK.

Вероятнее всего, вам следует декодировать строку в байты (т.е. в символы с кодовыми пунктами ниже 256) с помощью модуля Encode.

Классы символов

При поиске по шаблону можно искать любой символ, который обладает (или не обладает) определенным свойством. Существует четыре способа указать символьный класс. Традиционный способ – при помощи квадратных скобок и перечисления возможных символов, а также три способа на основе мнемонических сокращений: традиционные классы символов Perl, такие как \w, свойства Юникода, такие как \p{word}, или классы обратной совместимости с POSIX, такие как [:word:]. Каждое вхождение такого сокращения соответствует только одному символу из соответствующего набора. Снабдите их квантификаторами для поиска более длинных последовательностей. Так \d+ соответствует одной или нескольким цифрам. (Естественная ошибка – считать, что \w соответствует слову. Для поиска слова используйте \w+. Под «словом» понимается идентификатор языка программирования с подчеркиваниями и цифрами, а не слово на естественном языке.)

Классы символов в квадратных скобках

Перечисление символов, заключенное в квадратные скобки, называется *классом символов* и соответствует любому из символов этого списка (но только одному). Например, класс [aeiouy] соответствует любой гласной букве английского языка. Чтобы включить в класс правую квадратную скобку, поставьте перед ней обратную косую черту или сделайте ее первым элементом перечисления.

Диапазоны символов могут задаваться с помощью дефиса[1] в формате вида a-z. Можно объединять несколько диапазонов. Например, [0-9a-fA-F] соответствует одной шестнадцатеричной «цифре». Можно использовать обратную косую черту, чтобы экранировать дефис, который иначе будет интерпретироваться как указатель диапазона, либо поместить дефис в начале или конце класса (такой подход, возможно, снижает удобочитаемость, но является более традиционным).

Символ вставки ^ (или циркумфлекс, или «крышка», или «стрелка вверх») в начале перечисления символов инвертирует класс, в результате чего он соответствует любому символу, *не входящему* в список. (Чтобы включить в класс символ ^, *не делайте* его первым, а лучше попросту экранируйте его обратной косой чертой.) Например, [^aeiouy] соответствует любому символу, который не является гласной буквой английского языка. Однако будьте осторожны с отрицанием классов символов, поскольку вселенная символов расширяется. Например, этот класс символов соответствует согласным, а также пробелам, символам перевода строки и чему угодно (включая гласные) в кириллице, греческом и почти любом другом алфавите, не говоря уже обо всех китайских, японских и корейских иероглифах. А в один прекрасный день, может быть, даже в алфавитах Cirth и Tengwar[2]. (И уж наверняка в Линейном письме Б и этрусском.) Поэтому, возможно, будет лучше задать согласные явно, например, как [cbdfghjklmnpqrstvwxyz] или для краткости [b-df-hj-np-tv-z].

---

1 Фактически с помощью символа U+002D, HYPHEN-MINUS, но не U+2010, HYPHEN.

2 Cirth (Кирт, Кертар) и Tengwar (Тенгвар) – изобретенные Дж. Р. Р. Толкиеном два вида эльфийских алфавитов – рунический (Кертар) и буквенный (Тенгвар). – *Прим. ред.*

(Это также решает проблему с латинской «у», которая должна быть одновременно в обоих местах, что не получится при использовании ^ — дополнения класса.)

Внутри класса символов допускаются обычные символьные метазнаки, такие как \n, \t, \c*X*, \x*NN*, *NNN* (подразумевается восьмеричное число, а *не* обратная ссылка), \p{*YESPROP*} и \N{*NAME*}. Кроме того, в классе символов можно использовать \b для обозначения забоя, так же как в строках, заключенных в двойные кавычки. Обычно при поиске по шаблону \b обозначает границу слова. Но утверждения нулевой ширины не имеют смысла в классах символов, поэтому в них \b возвращается к своему обычному значению в строках. В качестве границы диапазона можно использовать любой одиночный символ, будь то литеральный символ, экранированная последовательность вроде \t, шестнадцатеричный или восьмеричный код символа, или же именованный символ.

В символьных классах также допускается использование любых метазнаков, представляющих определенные наборы символов, включая инвертированные классы вроде \P{NOPROP}, \N, \S и \D, а также предопределенные символьные классы, описываемые ниже в этой главе (традиционные, классы Юникода или классы POSIX). Но не пытайтесь применять их в качестве границ диапазона: это бессмысленно, поэтому символ «-» будет интерпретирован буквально. Не имеет также смысла использовать нечто, имеющее длину более одного символа. Это правило исключает \R, поскольку этот метазнак может соответствовать паре возврат каретки/ перевод строки; метазнак \X, поскольку он может соответствовать нескольким идущим подряд кодовым пунктам; а также некоторые именованные последовательности \N{*NAME*}, эквивалентные нескольким кодам символов.

Все остальные метазнаки утрачивают свое особое значение, находясь в квадратных скобках. В частности, не допускается применение трех обобщающих масок: ., \X и \C. Первое часто вызывает удивление, но не имеет смысла использовать универсальный класс символов внутри ограниченного, зато часто требуется найти точку в составе класса символов, например, при поиске имен файлов. Бессмысленно также задавать квантификаторы, утверждения или чередование внутри класса символов, поскольку символы интерпретируются индивидуально. Например, [fee|fie|foe|foo] означает то же самое, что и [feio|].

Символьный класс в квадратных скобках обычно соответствует только одному символу. По этой причине именованные последовательности Юникода не могут применяться (с пользой для дела) в классах символов в Perl v5.14. Они похожи на имена символов, но в действительности обозначают несколько символов. Например, LATIN CAPITAL LETTER A WITH MACRON AND GRAVE можно использовать в конструкции \N{...}, но фактически это имя раскрывается в символ U+0100, за которым следует символ U+0300. Внутри квадратных скобок эта именованная последовательность будет выглядеть как [\x{100}\x{300}], что навряд ли отвечает вашим желаниям.

Однако в области действия модификатора /i символьный класс иногда может соответствовать более чем одному символу. Это обусловлено тем, что в результате полной свертки регистра символов единственный символ в строке может соответствовать нескольким символам в шаблоне и наоборот. Например, следующее выражение истинно:

```
"SS" =~ /^[\xDF]$/iu
```

Причина в том, что результатом свертки регистра символа U+00DF является последовательность "ss", и результатом свертки регистра строки "SS" также является

последовательность "ss". Поскольку результаты свертки одинаковы, попытка сопоставления завершается успехом. Однако в инвертированных символьных классах, таких как [^\xDF], полная свертка заменяется простой сверткой, потому что иначе могут возникнуть логические противоречия. Это единственный случай, когда в Perl используется простая свертка регистра; во всех остальных случаях используется полная свертка и полное отображение регистров.

Классические сокращенные обозначения классов в Perl

С самого начала в Perl имелся ряд сокращенных обозначений классов символов. Они перечислены в табл. 5.11. Все они являются буквенными метазнаками, т.е. экранированы обратной косой чертой, и в каждом случае версия в верхнем регистре представляет собой отрицание версии в нижнем регистре.

Таблица 5.11. Классические классы символов

Символ	Значение	Обычное свойство	Свойство с модификатором /a	Перечисление с модификатором /a	Устаревший класс POSIX
\d	Цифра	\p{X_POSIX_Digit}	\p{POSIX_Digit}	[0-9]	[:digit:]
\D	Не цифра	\P{X_POSIX_Digit}	\P{POSIX_Digit}	[^0-9]	[:^digit:]
\w	Символ слова	\p{X_POSIX_Word}	\p{POSIX_Word}	[_A-Za-z0-9]	[:word:]
\W	Все, кроме символа слова	\P{X_POSIX_Word}	\P{POSIX_Word}	[^_A-Za-z0-9]	[:^word:]
\s	Пробельный символ	\p{X_Perl_Space}	\p{Perl_Space}	[\t\n\f\r]	[:space:][a]
\S	Непробельный символ	\P{X_Perl_Space}	\P{Perl_Space}	[^\t\n\f\r]	[:^space:]
\h	Горизонтальный пробельный символ	\p{Horiz_Space}	\p{Horiz_Space}	Много	[:blank:]
\H	Все, кроме горизонтального пробельного символа	\P{Horiz_Space}	\P{Horiz_Space}	Много	[:^blank:]
\v	Вертикальный пробельный символ	\p{Vert_Space}	\p{Vert_Space}	Много	–
\V	Все, кроме вертикального пробельного символа	\P{Vert_Space}	\P{Vert_Space}	Много	–

Круг их соответствия намного шире, чем можно себе представить, потому что обычно они обрабатывают полный диапазон символов Юникода, а не только ASCII (причем инвертированные классы могут даже выходить за пределы диапазона символов Юникода). В любом случае, соответствующие этим классам наборы сим-

[a] Но без VTAB.

волов обычно являются надмножеством набора ASCII или национальных алфавитов. Описание устаревших форм POSIX приводится в разделе «Классы символов в стиле POSIX» далее в этой главе. Чтобы сохранить старые значения байтов, можно использовать прагму use re "/a" в текущей области видимости или добавить в конкретный шаблон один или два модификатора /a.

(Да, мы в курсе, что большинство слов не содержит цифры и символы подчеркивания; \w под «словами» подразумевает лексемы типичного языка программирования. Скажем, Perl, если на то пошло.)

Эти метазнаки могут использоваться снаружи или внутри квадратных скобок как самостоятельно, так и в составе конструируемого класса символов:

```
if ($var =~ /\D/)        { warn "содержит нецифру" }
if ($var =~ /[^\w\s.]/) { warn "содержит не-(слово, пробел, точка)" }
```

Большинство из них имеют определения, соответствующие стандарту Юникода. И хотя внутренние механизмы Perl используют Юникод, существует множество старых программ, не подозревающих об этом, что может приводить к неприятным сюрпризам. Так, все традиционные символьные классы в Perl страдают расщеплением личности в том смысле, что иногда подразумевают одно, а иногда – другое. В области действия модификатора /u двойной режим работы с набором символов ASCII-Юникод отключается, и строки всегда интерпретируются, как последовательности символов Юникода. Так как это прямой путь к здравомыслию, данный режим используется по умолчанию в версиях Perl v5.14 и выше. (Особенность unicode_strings также включает режим работы с Юникодом по умолчанию.)

Исторически сложилось так, что класс \s отличается от [\h\v], потому что класс \v включает \cK – редко используемый символ вертикальной табуляции. По этой причине класс \s в языке Perl неточно соответствует свойству \p{Whitespace} Юникода.

Если вы пользуетесь старыми региональными настройками (употребляя use locale или use re "/l"), они действуют только для символов с кодовыми пунктами ниже 256, но для других символов действуют обычные правила Юникода.

При работе с кодами символов выше 255 Perl обычно переходит на исключительно посимвольную интерпретацию. Это означает, например, что код U+03A9, GREEK CAPITAL LETTER OMEGA, *всегда* будет соответствовать классу \w.

При использовании же модификатора /a или /aa все обстоит совсем иначе. Обычно эти модификаторы используются, чтобы обеспечить корректную работу старых шаблонов, созданных до появления Юникода. Чтобы не добавлять модификатор /a в каждый шаблон, где он необходим, можно воспользоваться следующей прагмой, имеющей лексическую область видимости и автоматически применяющей модификатор /a:

```
use re "/a";
```

Это правило исключает, например, некоторые пробельные символы. Прагма также подразумевает, что не-ASCII буквы из набора ISO-8859-1 больше не будут считаться символами, соответствующими классу \w.

Свойства символов

Свойства символов доступны посредством конструкции \p{PROP} и дополняющего множества \P{PROP}. Для семи основных свойств категорий Юникода, имена кото-

рых состоят из одного символа, фигурные скобки не обязательны. Так что можно использовать \pL для обозначения любой буквы или \pN для обозначения числа, но для всех остальных свойств, таких как \p{Lm} или \p{Nl}, фигурные скобки являются обязательными.

Большинство свойств определено непосредственно в стандарте Юникода, но некоторые (как правило, составные, определенные на основе стандартных свойств) являются специфическими для языка Perl. Например, Nl и Mn – это стандартные категории Юникода, представляющие буквенно-цифровые и непробельные комбинационные знаки, а Perl_Space – это собственное изобретение языка Perl.

Эти свойства можно использовать самостоятельно или объединять в пользовательском классе символов:

```
if ($var =~ /^\p{alpha}+$/) { say "одни буквы" }
if ($var =~ s/[^\pL\pN]//g)  { say "удалить все, кроме буквенно-цифровых" }
```

Свойств существует великое множество, и некоторые из наиболее часто используемых соответствуют гораздо большему количеству символов, чем можно было бы представить. Например, свойства alpha и word, каждое в отдельности, охватывают более 100 000 символов, причем свойство word, будучи надмножеством свойства alpha, является, как следствие, и более широким.

Текущий список свойств, поддерживаемых вашей версией Perl, включая количество символов, соответствующих каждому свойству, можно найти в странице *perluniprops* справочного руководства. Язык Perl внимательно следит за развитием стандарта Unicode Standard, поэтому, как только в Юникоде появляются новые свойства, они тут же добавляются в Perl. Официальный перечень свойств Юникода можно найти в приложении к стандарту Юникода «UAX #44: Unicode Character Database», а также в приложении C «Compatibility Properties» технического стандарта «UTS #18: Unicode Regular Expressions». Если существующих свойств окажется недостаточно, вы всегда сможете определить свои свойства – как это сделать, рассказывается в главе 6.

К числу наиболее часто используемых свойств принадлежат основные категории Юникода. В табл. 5.12 перечислены все семь категорий с односимвольными именами, а также длинные формы имен и значения.

Таблица 5.12. Основные категории Юникода (главные)

Краткое имя	Длинное имя	Значение
C	Other	Необычные управляющие и прочие символы
L	Letter	Буквы и идеограммы
M	Mark	Комбинационные знаки
N	Number	Числа
P	Punctuation	Знаки пунктуации
S	Symbol	Символы, знаки и обозначения
Z	Separator	Разделители (pazделители?)

Каждое из этих семи свойств в действительности является псевдонимом всех основных категорий с двухсимвольными именами, начинающимися с этой буквы. В табл. 5.13 приводится исчерпывающий (и замороженный) список всех основных

категорий. Все символы, даже еще не включенные в стандарт, принадлежат ровно одной категории из перечисленных ниже.

Таблица 5.13. Основные категории Юникода (все)

Краткое имя	Длинное имя	Значение
Cc	Control	Управляющие коды C0 и C1 из наборов ASCII и Latin-1
Cf	Format	Невидимые символы форматирования текста
Cn	Unassigned	Коды, которым пока не сопоставлены символы
Co	Private Use	Свободные для личного использования
Cs	Surrogate	Несимволы, зарезервированные для UTF-16
Ll	Lowercase_Letter	Маленькие символы
Lm	Modifier_Letter	Надстрочные символы и интервальные диакритические знаки
Lo	Other_Letter	Одноместные буквы и идеограммы
Lt	Titlecase_Letter	Заглавные буквы, используемые только в начале строки, например, в первом слове предложения
Lu	Uppercase_Letter	Прописные буквы
Mc	Spacing_Mark	Маленькие комбинационные знаки, занимающие отдельное место при выводе
Me	Enclosing_Mark	Комбинационные знаки, окружающие другие символы
Mn	Monospacing_Mark	Маленькие комбинационные знаки, не занимающие отдельное место при выводе
Nd	Decimal_Number	Цифры 0–9 для записи чисел в системе счисления с основанием 10
Nl	Letter_Number	Буквы, играющие роль цифр, такие как римские цифры
No	Other_Number	Любые другие числа, такие как дроби
Pc	Connector_Punctuation	Соединительные знаки, такие как подчеркивание
Pd	Dash_Punctuation	Любые дефисы и тире (но не «минус»)
Pe	Close_Punctuation	Знаки пунктуации, такие как закрывающие скобки
Pf	Final_Punctuation	Знаки пунктуации, такие как закрывающие кавычки
Pi	Initial_Punctuation	Знаки пунктуации, такие как открывающие кавычки
Po	Other_Punctuation	Все остальные знаки пунктуации
Ps	Open_Punctuation	Знаки пунктуации, такие как открывающие скобки
Sc	Currency_Symbol	Символы обозначения валют
Sk	Modifier_Symbol	В основном диакритические знаки
Sm	Math_Symbol	Математические символы
So	Other_Symbol	Все остальные символы
Zl	Line_Separator	Только U+2028
Zp	Paragraph_Separator	Только U+2029
Zs	Space_Separator	Все остальные неуправляющие пробельные символы

Все стандартные свойства Юникода фактически состоят из двух частей: \p{*NAME*=*VALUE*}. Поэтому все свойства, состоящие из одной части, являются дополнениями к официальным свойствам Юникода. Логические свойства со значением true всегда можно сократить до свойств, состоящих из одной части, что позволяет, например, записать свойство \p{Lowercase=True} как \p{Lowercase}. Свойства других типов, кроме логических, принимают строковые, числовые или перечислимые значения. В языке Perl имеются также псевдонимы для свойств всех основных категорий, алфавитов и блоков, состоящие из одной части, плюс реализуются рекомендации первого уровня из технического стандарта Юникода «Unicode Technical Standard #18, Regular Expressions» (версии 13, от 2008-08), такие как \p{Any}.

Например, \p{Armenian}, \p{IsArmenian} и \p{Script=Armenian} – это одно и то же свойство, так же как \p{Lu}, \p{GC=Lu}, \p{Uppercase_Letter} и \p{General_Category=Uppercase_Letter}. В качестве других примеров логических свойств (неявно имеющих истинные значения) можно привести \p{Whitespace}, \p{Alphabetic}, \p{Math} и \p{Dash}. Примеры свойств других типов: \p{Bidi_Class=Right_to_Left}, \p{Word_Break=A_Letter} и \p{Numeric_Value=10}. На странице справочного руководства *perluniprops* перечислены все свойства и их псевдонимы в Perl, как стандартные, так и специфические для Perl.

Результат сопоставления символа, не являющегося символом Юникода (т.е. с кодом выше 0x10FFFF), со свойством Юникода, не определен. Сейчас, по умолчанию, Perl выдает предупреждение, а сопоставление завершается безрезультатно. В некоторых случаях такое поведение может выглядеть довольно странно:

```
chr(0x110000) =~ /\p{ahex=true}/  # false
chr(0x110000) =~ /\p{ahex=false}/ # false!

chr(0x110000) =~ /\P{ahex=true}/  # true
chr(0x110000) =~ /\P{ahex=false}/ # true!
```

Однако свойства, определяемые пользователем, можно наделить любым поведением. См. раздел «Конструирование символа» в главе 6.

Классы символов в стиле POSIX

В отличие от других сокращенных обозначений классов символов в Perl, обозначения классов символов в стиле POSIX, имеющие вид [:*CLASS*:], могут использоваться *только* при создании других классов символов, т.е. внутри другой пары квадратных скобок. Например /[.,[:alpha:][:digit:]]/ осуществляет поиск одного символа, который является литеральной точкой (поскольку она находится в классе символов), или запятой, или буквой алфавита, или цифрой. Для этой цели также можно использовать свойства символов с теми же именами, например, [.,\p{alpha}\p{digit}].

За исключением свойства punct, о котором речь пойдет в следующем абзаце, имена классов в стиле POSIX можно применять в свойствах вида \p{} и \P{}, обладающих тем же смыслом. У такого подхода два преимущества: во-первых, свойства проще в наборе, ведь их не требуется окружать дополнительными квадратными скобками; во-вторых – что, вероятно, более важно, – благодаря тому, что свойства не подвержены действию модификаторов кодовых страниц, они всегда соответствуют символам Юникода. В то же время запись вида [[:...:]] для классов POSIX, *напротив*, делает их подконтрольными флагам модификаторов.

Свойство \p{punct} отличается от POSIX-класса [[:punct:]] тем, что \p{punct} никогда не соответствует символам, не являющимся знаками пунктуации, а класс [[:punct:]] (и свойства \p{POSIX_Punct} и \p{X_POSIX_Punct}) соответствует. Это обусловлено тем, что стандарт Юникода разбивает единую POSIX-группу знаков пунктуации на две категории: знаки пунктуации и символы. В отличие от \p{punct}, другие свойства, упомянутые выше, также будут соответствовать символам, перечисленным в табл. 5.14.

Таблица 5.14. ASCII-символы, интерпретируемые как знаки пунктуации

Глиф	Код	Категория	Алфавит	Название
$	U+0024	GC=Sc	SC=Common	DOLLAR SIGN
+	U+002B	GC=Sm	SC=Common	PLUS SIGN
<	U+003C	GC=Sm	SC=Common	LESS-THAN SIGN
=	U+003D	GC=Sm	SC=Common	EQUALS SIGN
>	U+003E	GC=Sm	SC=Common	GREATER-THAN SIGN
^	U+005E	GC=Sk	SC=Common	CIRCUMFLEX ACCENT
`	U+0060	GC=Sk	SC=Common	GRAVE ACCENT
|	U+007C	GC=Sm	SC=Common	VERTICAL LINE
~	U+007E	GC=Sm	SC=Common	TILDE

Можно считать, что класс [[:punct:]] соответствует всем символам, которые в Юникоде считаются знаками пунктуации (когда действуют правила интерпретации символов Юникода), плюс еще девяти знакам пунктуации из диапазона ASCII, перечисленным в табл. 5.14, которые в Юникоде считаются символами.

В первой колонке табл. 5.15 приведены классы POSIX, доступные в v5.14.

Таблица 5.15. Классы символов POSIX

Класс	Обычное значение	Значение с модификатором /a*
alnum	Любой алфавитно-цифровой символ, т.е. любой символ alpha или digit. В том числе множество символов, не являющихся буквами – см. следующую строку таблицы. Эквивалентен свойству \p{X_POSIX_Alnum}.	Строго [A-Za-z0-9]. Эквивалентен свойству \p{POSIX_Alnum}.*
alpha	Любой алфавитный символ вообще, включая все буквы, плюс другие символы со свойством Other_Alphabetic, такие как римские цифры, символы букв в кружочках и комбинационный знак «йота» греческого алфавита. Эквивалентен свойству \p{X_POSIX_Alpha}.	Строго 52 ASCII-символа [A-Za-z]. Эквивалентен свойству \p{POSIX_Alpha}.*
ascii	Только символы с числовыми кодами 0–127. Эквивалентен свойству \p{ASCII}.	Любой символ с числовым кодом 0–127. Эквивалентен свойству \p{ASCII}.*
blank	Любой горизонтальный пробельный символ. Эквивалентен свойствам \p{X_POSIX_Blank} и \p{HorizSpace}, а также \h.	Только пробел или табуляция. Эквивалентен свойству \p{POSIX_Blank}.

Класс	Обычное значение	Значение с модификатором /a♣
cntrl	Любой символ со свойством Control. Обычно символы из этого класса не генерируют вывод как таковой, но оказывают некоторое управляющее воздействие на терминал. Например, они отвечают за переводы строк, разрыв страниц и забой. В это множество в настоящее время входят все символы с кодовыми пунктами 0–31 и 127–159. Эквивалентен свойству \p{X_POSIX_Cntrl}.	Любые символы с кодовыми пунктами 0–31 и выше 127. Эквивалентен свойству \p{POSIX_Cntrl}.
digit	Любой символ со свойством Digit. Технически говоря, этом символы со свойством Numeric_Type=Decimal, занимающие непрерывные диапазоны длиной 10 символов, числовые значения которых последовательно возрастают от 0 до 9 (Numeric_Value=0..9). Эквивалентен свойству \p{X_POSIX_Digit} или \d.	10 символов от «0» до «9». Эквивалентен свойству \p{POSIX_Digit} или \d с модификатором /a.
graph	Любой не-Whitespace символ, не входящий в категории Control, Surrogate и Unassigned. Эквивалентен свойству \p{X_POSIX_Graph}.	Множество ASCII-символов, минус пробельные и управляющие, т.е. любые символы с числовыми кодами в диапазоне 33–126. Эквивалентен свойству \p{POSIX_Graph}.♣
lower	Любой символ нижнего регистра, не только буквы. Включает все символы из категории Lowercase_Letter, плюс символы со свойством Other_Lowercase. Эквивалентен свойству \p{X_POSIX_Lower} или \p{Lowercase}. С модификатором /i также соответствует любому символу с GC=LC, сокращенному обозначению любого из GC=Lu, GC=Lt и GC=Ll.	Только 26 букв ASCII в нижнем регистре [a-z]. С модификатором /i также включает [A-Z]. Эквивалентен свойству \p{POSIX_Lower}.♣
print	Любой символ из класса graph или не входящий в классы cntrl и blank. Эквивалентен свойству \p{X_POSIX_Print}.	Любой символ из класса graph или не входящий в классы cntrl и blank. Эквивалентен свойству \p{POSIX_Print}.♣
punct	Любой символ из категории Punctuation, плюс девять ASCII-символов из категории Symbol. Эквивалентен свойству \p{X_POSIX_Punct} или \pP.	Любой ASCII-символ из категории Punctuation или Symbol. Эквивалентен свойству \p{POSIX_Punct}
space	Любой символ со свойством Whitespace, включая табуляцию, перевод строки, вертикальную табуляцию, перевод страницы, возврат каретки, пробел, неразрывный пробел, следующая строка, узкий пробел, тонкая шпация, разделитель абзацев Юникода и еще великое множество других. Эквивалентен свойству \p{X_POSIX_Space}, [\v\h] или [\s\cK]; класс \s сам по себе эквивалентен свойству \p{X_Perl_Space}, если из последнего выбросить соответствие \cK, т.е. вертикальную табуляцию.	Любой ASCII-символ со свойством Whitespace, включая табуляцию, перевод строки, вертикальную табуляцию, перевод страницы, возврат каретки и пробел. Эквивалентен свойству \p{POSIX_Space}; в самом классе \s отсутствует вертикальная табуляция.

Таблица 5.15 (продолжение)

Класс	Обычное значение	Значение с модификатором /a*
upper	Любой символ верхнего (не заглавного) регистра, не только буквы. Включает все символы из категории Uppercase_Letter, плюс символы со свойством Other_Uppercase. С модификатором /i также соответствует любому символу, в котором верхний регистр совпадает с результатом свертки. Эквивалентен свойству \p{X_POSIX_Upper} или \p{Uppercase}.	Только 26 букв ASCII в верхнем регистре [A-Z]. С модификатором /i также включает [a-z]. Эквивалентен свойству \p{POSIX_Upper}.*
word	Любой символ из класса alnum и из категорий Mark и Connector_Punctuation. Эквивалентен свойству \p{X_POSIX_Word} или \w. Обратите внимание, что идентификаторы с символами Юникода, в том числе и в языке Perl, следуют собственным правилам: первый символ должен иметь свойство ID_Start, а все последующие – свойство ID_Continue. (Язык Perl также допускает в качестве первого символа идентификатора символ со свойством Connector_Punctuation.)	Любая буква ASCII, цифра или символ подчеркивания. Эквивалентен свойству \p{POSIX_Word} или \w с модификатором /a.*
xdigit	Любая шестнадцатеричная цифра, либо однобайтные символы ASCII, либо соответствующие многобайтные символы. Эквивалентен свойству \p{X_POSIX_XDigit}, \p{Hex_Digit} или \p{hex}.	Любая шестнадцатеричная цифра из набора ASCII. Эквивалентен классу [0-9A-Fa-f], свойству \p{POSIX_XDigit}, \p{ASCII_Hex_Digit} или \p{ahex}.

Классы из таблицы, помеченные символом *, могут также соответствовать некоторым не-ASCII символам при использовании модификаторов /ai. В настоящее время к таковым относятся:

```
ſ  U+017F GC=Ll SC=Latin    LATIN SMALL LETTER LONG S
K  U+212A GC=Lu SC=Latin    KELVIN SIGN
```

Первый символ при свертке регистра преобразуется в обычный символ «s» нижнего регистра, а второй – в обычный символ «k» нижнего регистра. Имеется возможность подавить такое поведение, продублировав модификатор /a: /aai.

Можно создать отрицание (или дополнение) POSIX-класса символов, поместив перед именем класса и сразу после [: символ ^ (это расширение Perl), как показано в табл. 5.16.

Таблица 5.16. POSIX-классы символов и их эквиваленты в Perl

POSIX	Классический
[:^digit:]	\D
[:^space:]	\S
[:^word:]	\W

Квадратные скобки являются частью конструкции класса символов в стиле POSIX [::], а не частью класса символов в целом. Поэтому можно писать такие

шаблоны, как /^[[:lower:][:digit:]]+$/ для поиска строки, состоящей только из букв в нижнем регистре или цифр (и, возможно, замыкающего символа перевода строки). В частности, следующая команда не работает:

```
42 =~ /^[:digit:]$/            # НЕВЕРНО
```

Это неверно, потому что POSIX-класс здесь находится не внутри класса символов. Но *это класс символов*, представляющий символы «:», «i», «t», «g» и «d». Perl безразлично, что символ «:» задан дважды.

Вот что нужно было написать в действительности:

```
42 =~ /^[[:digit:]]+$/
```

POSIX-классы символов [.cc.] и [=cc=] распознаются, но порождают ошибку, в которой сообщается, что они не поддерживаются.

Квантификаторы

Если не указано иное, то для каждого элемента в регулярном выражении отыскивается только одно соответствие. Для такого шаблона, как /nop/, должны быть найдены все эти символы, один за другим. Такие слова, как «panoply» и «xenophobia» соответствуют шаблону, потому что не имеет значения, *где* именно в строке расположено соответствие.

Если необходимо обеспечить соответствие пары слов «xenophobia» и «Snoopy», то шаблон /nop/ использовать нельзя, поскольку он требует наличия только одной буквы «o» между «n» и «p», а в слове «Snoopy» их две. В таких случаях на помощь приходят *квантификаторы*: они сообщают, сколько раз что-то должно присутствовать, тогда как по умолчанию ищется лишь одно соответствие. Квантификаторы в регулярном выражении – как циклы в программе. В действительности, если представить себе регулярное выражение как программу, они и *есть* циклы. Некоторые циклы точные, типа «повторить это соответствие ровно пять раз» ({5}). Другие задают нижнюю и верхнюю границы счетчика соответствий, типа «повторить это соответствие не меньше двух, но не больше четырех раз» ({2,4}). Некоторые вообще не определяют верхнюю границу, вроде «не менее двух соответствий, но сколь угодно много» ({2,}).

В табл. 5.17 перечислены квантификаторы, которые Perl распознает в шаблоне.

Таблица 5.17. Квантификаторы в регулярных выражениях

Максимальный	Минимальный	Неуступающий	Допустимый диапазон
{*MIN*,*MAX*}	{*MIN*,*MAX*}?	{*MIN*,*MAX*}?+	Не меньше *MIN* соответствий, но не больше *MAX*
{*MIN*,}	{*MIN*,}?	{*MIN*,}?+	Не меньше *MIN* соответствий
{*COUNT*}	{*COUNT*}?	{*COUNT*}?+	Ровно *COUNT* соответствий
*	*?	*+	0 или более соответствий (то же, что и {0,})
+	+?	++	1 или более соответствий (то же, что и {1,})
?	??	?+	0 или 1 соответствие (то же, что и {0,1})

Конструкция, сопровождаемая квантификатором * или ?, фактически не обязана иметь соответствие в строке. Дело в том, что соответствие может произойти 0 раз, что будет засчитано за положительный результат. Квантификатор + часто подходит лучше, поскольку предполагает хотя бы одно соответствие.

Пусть вас не смущает использование в предыдущей таблице слова «ровно». Оно относится только к счетчику повторений, а не к строке в целом. Например, `$n =~ /\d{3}/` не означает: «Равна ли длина строки ровно трем цифрам?» Здесь спрашивается, есть ли в `$n` место, где подряд идут три цифры. Строка `"101 Morris Street"` удовлетворяет этому, как и строки `"95472"` и `"1-800-555-1212"`. Все они *содержат* три цифры в одной или нескольких позициях, и это все, о чем мы спрашивали. Для закрепления прочитайте раздел «Позиции» далее в этой главе, где описывается использование позиционных утверждений (как в `/^\d{3}$/`).

Имея возможность найти варьирующееся число соответствий, максимальные квантификаторы выбирают максимальное число повторений. Поэтому, когда мы говорим: «Столько раз, сколько угодно», жадный квантификатор интерпретирует это как «сколько сможешь унести», ограничиваясь лишь требованием, чтобы это не помешало поиску соответствий, согласно оставшимся спецификациям шаблона. Если в шаблоне есть два неограниченных квантификатора, то, очевидно, оба они не могут «съесть» строку целиком: символы, которые использует одна часть соответствия, становятся недоступными части, следующей за ней. В этом случае квантификаторы демонстрируют жадность, обделяя следующие за ними, при этом шаблон читается слева направо.

Это привычное поведение квантификаторов в регулярных выражениях. Однако Perl позволяет нам изменить поведение своих квантификаторов: поместив ? после квантификатора, мы превратим его из максимального в минимальный. Это не значит, что минимальный квантификатор всегда будет давать соответствие, состоящее из минимального числа повторений, разрешенных в его диапазоне, как не означает и того, что максимальный квантификатор всегда будет находить максимальное число соответствий, допустимое в его диапазоне. Успешным должно быть соответствие шаблону в целом, и минимальное соответствие возьмет столько, сколько нужно, чтобы быть успешным, и не более. (Минимальные квантификаторы удовлетворение ценят выше жадности.)

Например, при поиске

```
"exasperate" =~ /e(.*)e/        # $1 содержит теперь "xasperat"
```

подвыражение `.*` соответствует подстроке `"xasperat"` – самой длинной из возможных. (Это значение также записывается в `$1`, как будет видно из раздела «Захват и группировка» далее в этой главе.) Хотя имелось более короткое соответствие, при жадном поиске это не важно. Если в одной и той же точке есть возможность выбора, всегда возвращается *более длинный* вариант.

Сравните это со следующим:

```
"exasperate" =~ /e(.*?)e/       # $1 содержит теперь "xasp"
```

Здесь применяется минимальная версия поиска, `.*?`. Добавление ? к * заставляет `*?` вести себя противоположным образом: теперь, если в одной точке есть выбор из двух возможностей, всегда возвращается *более короткая* из них.

Хотя можно прочесть `*?` как «искать ноль или более чего-то, но предпочтительно ноль», это не значит, что всегда будет отыскиваться ноль символов. Будь это так,

и в нашем примере $1 была установлена в "", не нашлась бы вторая "e", поскольку она не следует за первой непосредственно.

Вы можете поинтересоваться, почему при минимальном поиске /e(.*?)e/ Perl не поместил в переменную $1 строку "rat". Ведь "rat" тоже находится между двумя "e" и короче "xasp". В Perl выбор между максимальным и минимальным происходит только при наличии альтернатив соответствия, причем когда все они имеют одну и ту же начальную точку. Если есть два возможных соответствия, но начинаются они на разном смещении от начала строки, их длины не играют роли, как и то, какой квантификатор используется – минимальный или максимальный. Из нескольких возможных соответствий побеждает всегда то, которое встретилось раньше. Лишь если несколько возможных соответствий начинаются в одной и той же точке, для принятия решения используется минимальное или максимальное соответствие. А если начальные точки различаются, то и решать нечего. Обычно Perl ищет *самое левое длиннейшее* соответствие; при минимальном поиске оно становится *самым левым кратчайшим*. Часть «самое левое» всегда постоянна и является доминирующим критерием.[1]

Есть два способа преодолеть «левый уклон» поиска по шаблону. Во-первых, можно использовать сначала жадный квантификатор (обычно .*), чтобы постараться «проглотить» начальные части строки. При поиске соответствия для жадного квантификатора сначала отыскивается самое длинное соответствие, в результате чего оставшаяся часть строки просматривается справа налево:

```
"exasperate" =~ /.*e(.*?)e/    # $1 теперь содержит "rat"
```

Но будьте осторожны, поскольку общий результат поиска теперь содержит всю строку вплоть до этой точки.

Вторым способом преодоления стремления влево является использование позиционных утверждений, обсуждаемых в следующем разделе.

Как можно изменить максимальное поведение квантификатора на минимальное, добавив после него ?, точно так же максимальное поведение можно изменить на неуступающее, добавив после квантификатора +. Неуступающие квантификаторы позволяют управлять механизмом возвратов. Оба квантификатора, минимальный и максимальный, всегда осуществляют перебор всех возможных комбинаций при поиске соответствия. Неуступающий квантификатор никогда не уступит уже совпавшие символы, чтобы позволить найти другую возможную комбинацию, за счет чего способен существенно повысить скорость сопоставления.

С простыми сопоставлениями проблема производительности возникает нечасто, но с увеличением количества квантификаторов, особенно вложенных, она начинает приобретать все большее значение. Обычно применение неуступающих квантификаторов не влияет на общий успех сопоставления, если он возможен, но если сопоставление должно завершиться безрезультатно, они позволят достичь отрицательного результата быстрее. Например, такое сопоставление:

---

[1] Не все механизмы обработки регулярных выражений действуют таким образом. Некоторые веруют во всепоглощающую жадность, при которой всегда побеждает самое длинное соответствие, даже если оно находится дальше от начала строки. Perl не таков. Можно сказать, что рвение сильнее жадности (или расчетливости). Более формальное обсуждение этого и многих других принципов можно найти в разделе «Маленький Механизм, который /(не)? может/».

```
("a" x 20   "b") =~ /(a*a*a*a*a*a*a*a*a*a*a*a*)*[^Bb]$/
```

в конечном счете, не даст результатов. Но механизму регулярных выражений придется немало потрудиться, чтобы опробовать все возможные комбинации, допускаемые квантификаторами. Он не понимает, что поиск в данном случае обречен на неудачу. Изменив один или несколько максимальных квантификаторов * на неуступающий *+, можно заставить поиск завершиться намного быстрее. В данном случае, изменив последний максимальный квантификатор «звездочка» на неуступающий, можно получить неудачный результат раз в сто быстрее – достаточно серьезный прирост скорости.

Безусловно, это достаточно надуманный пример, но при создании сложных шаблонов подобная необходимость может возникнуть совершенно естественным путем. Оказывается, неуступающие квантификаторы действуют точно так же, как неуступающие группы, с которыми мы познакомимся ниже. Выражение a*+ с неуступающим квантификатором эквивалентно выражению (?>a*). Неуступающие группы обеспечивают чуть больше гибкости, чем неуступающие квантификаторы, потому что позволяют объединять совпавшие фрагменты в неделимые группы, недоступные для механизма возвратов. Но неуступающие квантификаторы проще набирать с клавиатуры, и зачастую их вполне достаточно, чтобы предотвратить катастрофическую деградацию производительности, обусловленную работой механизма возвратов.

Позиции

Некоторые конструкции в регулярных выражениях символизируют *позиции* в строке, которые должны быть найдены. Позиция – это место слева или справа от фактического символа. Эти метазнаки являются примерами утверждений *нулевой ширины*, поскольку они не соответствуют фактическим символам строки. Часто мы называем их просто утверждениями (assertions). (Они называются также точками привязки, или якорями (anchors), поскольку привязывают какую-то часть шаблона к определенной позиции.)

Есть возможность манипулировать позициями в строке без использования шаблонов. Встроенная функция substr позволяет извлекать подстроки и осуществлять присваивание подстрокам, измеряемым от начала строки, от конца строки или с конкретного числового смещения. Этого может оказаться достаточно при работе, например, с записями фиксированной длины. Шаблоны необходимы, только если числового смещения недостаточно. Но чаще всего смещений недостаточно или, по крайней мере, они недостаточно удобны в сравнении с шаблонами.

Начало строки: утверждения \A и ^

Утверждение \A соответствует только началу хранящегося в переменной текста, каким бы она ни был. Утверждение же ^ всегда соответствует началу хранящегося в переменной текста, но может соответствовать и началу любой строки, которую этот текст содержит. Если в шаблоне используется модификатор /m, а текст содержит внедренные символы перевода строки, ^ будет также соответствовать любому месту в строке сразу за символом перевода строки:

```
/\Abar/               # Соответствует "bar" и "barstool"
/^bar/                # Соответствует "bar" и "barstool"
/^bar/m               # Соответствует "bar" и "barstool" и "sand\nbar"
```

В сочетании с модификатором /g модификатор /m позволяет находить соответствие метасимволу ^ многократно в одной и той же строке:

```
s/^\s+//gm;           # Обрезать ведущие пробельные символы во всех строчках
$total++ while /^./mg; # Подсчитать непустые строчки
```

Конец строки: утверждения \z, \Z и $

Метазнак \z соответствует концу текста независимо от ее содержания. \Z соответствует позиции перед символом перевода строки в конце текста, если этот символ там есть, и позиции в конце текста, если такого символа нет. Метасимвол $ обычно имеет тот же смысл, что и \Z. Однако с модификатором /m, при наличии в тексте символов перевода строки, $ может соответствовать любому месту в тексте перед символом перевода строки:

```
/bot\z/        # Соответствует "robot"
/bot\Z/        # Соответствует "robot" и "abbot\n"
/bot$/         # Соответствует "robot" и "abbot\n"
/bot$/m        # Соответствует "robot" и "abbot\n" и "robot\nrules"

/^robot$/      # Соответствует "robot" и "robot\n"
/^robot$/m     # Соответствует "robot" и "robot\n" и "this\nrobot\n"
/\Arobot\Z/    # Соответствует "robot" и "robot\n"
/\Arobot\z/    # Соответствует только "robot" - почему бы не использовть eq?
```

Как и для ^, модификатор /m позволяет находить соответствие $ многократно в одном и том же тексте, если задан модификатор /g. (В следующих примерах предполагается, что в $_ считана запись из нескольких строк; возможно, перед чтением придется присвоить переменной $/ пустую строку "".)

```
s/\s*$//gm;           # Обрезать замыкающие пробелы во всех строках абзаца

while (/^([^:]+):\s*(.*)/gm ) { # получить заголовок почтового сообщения
    $headers{$1} = $2;
}
```

В разделе «Интерполяция переменных», далее в этой главе, мы расскажем, как интерполировать переменные в шаблонах: если $foo имеет значение "bc", то выражение /a$foo/ эквивалентно выражению /abc/. Здесь $ не соответствует концу строки. Чтобы метасимвол $ соответствовал концу строки, он должен быть в конце шаблона, либо сразу за ним должна следовать вертикальная черта или закрывающая скобка.

Границы: утверждения \b и \B

Утверждение \b соответствует любой границе слова, определяемой как позиция между символом \w и символом \W, в любом порядке. Порядок \W\w соответствует границе начала слова, а порядок \w\W – границе конца слова. (Концы строки рас-

сматриваются здесь как символы \W.) Утверждение \B соответствует любой позиции, которая не является границей слова, т.е. находится в середине \w\w или \W\W.

```
/\bis\b/        # Соответствует "what it is" и "that is it"
/\Bis\B/        # Соответствует "thistle" и "artist"
/\bis\B/        # Соответствует "istanbul" и "so-isn't that butter?
/\Bis\b/        # Соответствует "confutatis" и "metropolis near you"
```

Поскольку \W включает все знаки пунктуации (кроме подчеркивания), границы \b присутствуют в середине таких строк, как "isn't", "booktech@oreilly.com", "M.I.T." и "key/value".

Внутри класса символов ([\b]) знак \b представляет символ забоя, а не границу слова.

Последовательный поиск

В области действия модификатора /g функция pos позволяет читать или устанавливать смещение для следующей попытки поиска:

```
$burglar = "Bilbo Baggins";
while ($burglar =~ /b/gi) {
    printf "Найдена 'B' в позиции %d\n", pos($burglar)-1;
}
```

(Мы вычитаем из позиции единицу, поскольку ищем длину строки, а pos всегда указывает на позицию после найденного соответствия.)

Вышеприведенный код выдает:

```
Найдена 'B' в позиции 0
Найдена 'B' в позиции 3
Найдена 'B' в позиции 6
```

После безрезультатного поиска позиция обычно обнуляется. Если дополнительно указать модификатор /c (от слова «continue», продолжить), то, когда очередной поиск с модификатором /g окажется неудачным, указатель позиции не будет обнулен. Это позволяет продолжить поиск с текущей точки, а не начинать его с самого начала.

```
$burglar = "Bilbo Baggins";
while ($burglar =~ /b/gci) {    # ADD /c
    printf "Найдена 'B' в позиции %d\n", pos($burglar)-1;
}
while ($burglar =~ /i/gi) {
    printf "Найдена 'I' в позиции %d\n", pos($burglar)-1;
}
```

Помимо трех B, найденных ранее, Perl сообщает теперь об i, найденной в позиции 10. Без /c вторая попытка поиска в цикле началась бы с начала и была бы обнаружена еще одна i в позиции 1.

Где вы остановились: утверждение \G

При мысли об использовании функции pos возникает соблазн разделить строку с помощью функции substr, но такой подход редко бывает правильным. Чаще, если

уж мы начали с поиска по шаблону, то лучше так и продолжить. Но если вы озадачились позиционным утверждением, то, вероятно, вам нужен метасимвол \G.

Утверждение \G представляет в спецификации шаблона ту же точку, которую функция pos представляет за ее пределами. Если мы осуществляем последовательный поиск в строке с модификатором /g (или воспользовались функцией pos для непосредственного выбора начальной точки), то можем использовать \G для определения позиции сразу за предыдущим найденным соответствием. Иначе говоря, это утверждение соответствует месту непосредственно перед тем символом, текущая позиция которого будет определена посредством pos. Это позволяет запомнить, где мы остановились:

```
($recipe = <<'DISH') =~ s/^\s+//gm;
    Preheat oven to 451 deg. fahrenheit
    Mix 1 ml. dilithium with 3 oz. NaCl and
    stir in 4 anchovies. Glaze with 1 g.
    mercury. Heat for 4 hours and let cool
    for 3 seconds. Serves 10 aliens.
DISH

$recipe =~ /\d+ /g;
$recipe =~ /\G(\w+)/;  # $1 теперь "deg"
$recipe =~ /\d+ /g;
$recipe =~ /\G(\w+)/;  # $1 теперь "ml"
$recipe =~ /\d+ /g;
$recipe =~ /\G(\w+)/;  # $1 теперь "oz"
```

Метасимвол \G часто применяется в цикле, как показано в следующем примере. Мы «останавливаемся» после каждой цифровой последовательности и проверяем, нет ли за ней аббревиатуры. Если есть, захватываем два следующих слова, в противном случае только одно следующее слово:

```
pos($recipe) = 0;                      # На всякий случай сбросим \G в 0
$recipe =~ /(\d+) /g ) {
    my $amount = $1;
    if ($recipe =~ / \G (\w{0,3}) \. \s+ (\w+) /x) { # сокращ. + слово
        say "$amount $1 of $2";
    } else {
        $recipe =~ / \G (\w+) /x;      # просто слово
        say "$amount $1";
    }
}
```

В результате получается:

```
451 deg of fahrenheit
1 ml of dilithium
3 oz of NaCl
4 anchovies
1 g of mercury
4 hours
3 seconds
10 aliens
```

Захват и группировка

Существует возможность части шаблонов в подшаблоны и запоминать строки, соответствующие этим подшаблонам. Первое мы называем *группировкой* (*grouping*), а второе – *захватом* (*capturing*). Группировку можно производить и без захвата, но об этом мы поговорим позже.

Захват

Чтобы сохранить подстроку для дальнейшего ее использования, заключите в круглые скобки соответствующую этой подстроке часть шаблона. Первая пара скобок сохраняет соответствующую подстроку в переменной $1, вторая пара – в $2 и т. д. Скобок может быть сколько угодно; Perl создает нужное количество нумерованных переменных, представляющие захваченные строки.

Некоторые примеры:

```
/(\d)(\d)/    # Найти две цифры, захватив их в $1 и $2
/(\d+)/       # Найти одну или более цифр, захватив их все в $1
/(\d)+/       # Найти цифру один или более раз, захватив последнюю в $1
```

Обратите внимание на различия между вторым и третьим шаблонами. Обычно требуется то, что делает вторая форма. Третья форма *не* создает несколько переменных для нескольких цифр. Скобки нумеруются на этапе компиляции шаблона, а не на этапе поиска соответствий.

Захваченные строки часто называют *ссылками на группы* (*group references*), поскольку они ссылаются на части захваченного текста. Исторически сложилось так, что механизмы сопоставления с шаблоном ограничивают возможность ссылаться на группы исключительно *обратными ссылками* (*backreferences*), но в Perl можно ссылаться на любые группы, как находящиеся позади текущей позиции в шаблоне, так и впереди, и даже на текущую группу.

Существует два способа получить эти ссылки. Нумерованные переменные, которые мы только что описали, позволяют получать обратные ссылки за пределами шаблона, но в самом шаблоне они не работают. В шаблоне же приходится использовать нотацию обратных ссылок: \1, \2, \g{1}, \g{2}, \k<some_group>, \k<other_group> и т. д.

Выражение вроде $1 не позволяет сослаться на группу внутри шаблона, потому к моменту компиляции регулярного выражения эта конструкция уже будет интерполирована как обычная переменная. Поэтому внутри шаблонов следует использовать традиционную запись \1. Двухзначные и трехзначные номера обратных ссылок создают неоднозначность, поскольку такая форма записи совпадает с восьмеричной формой определения символов. Perl элегантно решает эту проблему, ориентируясь на количество сохраняющих групп в шаблоне. Например, если Механизм обнаруживает метасимвол \11, то считает его эквивалентом ссылки $11, только если прежде в шаблоне будет сохранено как минимум 11 подстрок. В противном случае значение интерпретируется как \011 – т.е. как символ табуляции. Чтобы избежать подобной неоднозначности, для ссылок на сохраняющие группы используйте форму \g{*NUMBER*}, а для определения кодов символов в восьмеричном виде – форму \o{*OCTNUM*}. Благодаря этому \g{11} всегда будет интерпретироваться как ссылка на 11-ю сохраняющую группу, а \o{11} – как символ с кодовым

пунктом 11 в восьмеричной системе. Однако еще лучше использовать именованные группы, как описывается ниже.

Итак, чтобы отыскать повторяющиеся слова, такие как "the the" или "had had", можно использовать следующий шаблон:

```
/\b(\w+) \1\b/i
```

Но чаще всего вы будете прибегать к форме $1, потому что обычно после применения шаблона выполняются некоторые действия с найденными подстроками. Допустим, имеется некоторый текст (почтовый заголовок), который выглядит так:

```
From: gnat@perl.com
To: camelot@oreilly.com
Date: Mon, 17 Jul 2011 09:00:00 _1000
Subject  Eye of the needle
```

и требуется создать хеш, отображающий текст перед каждым двоеточием в текст после него. Если организовать построчный перебор этого текста в цикле (например, при чтении из файла), создать хеш можно так:

```
while (<>) {
    /^(.*?): (.*)$/;    # Текст до двоеточия - в $1, после него - в $2
    $fields{$1} = $2
}
```

Подобно $`, $& и $' эти нумерованные переменные имеют динамическую область видимости, простирающуюся до конца охватывающего блока или строки eval либо до следующего успешного поиска по шаблону, в зависимости от того, что встретится раньше. Нумерованные переменные можно также использовать в правой (заменяющей) части операции подстановки:

```
s/^(\S+) (\S+)/$2 $1/; # Поменять местами первые два слова
```

Группы могут быть вложенным, и в этом случае нумерация групп производится согласно положению левой скобки. Поэтому для строки «Primula Brandybuck» шаблон

```
/^((\w+) (\w+))$/
```

захватит "Primula Brandybuck" в $1, "Primula" в $2 и "Brandybuck" в $3. Это отражено на рис. 5.1.

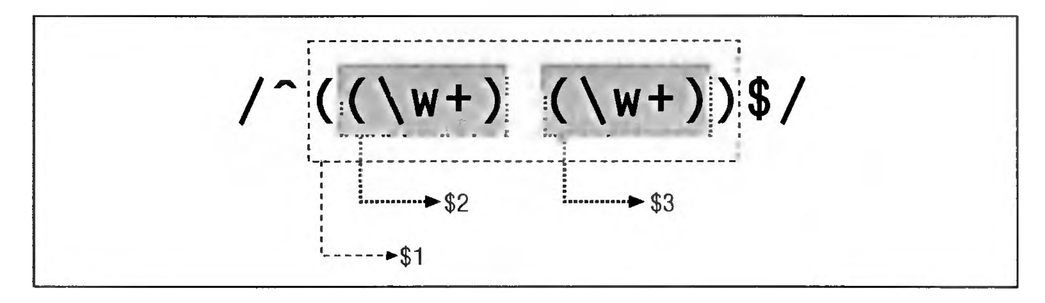


Рис. 5.1. *Создание ссылок на найденный текст с помощью круглых скобок*

Как упоминалось выше, ссылки на группы не обязаны быть обратными ссылками. Допускается ссылаться на любые группы, имеющиеся в шаблоне, даже если они еще не были сопоставлены тексту. Эта возможность полезна только для многократных обращений к одной и той же ссылке на группу. Первое обращение к опережающей ссылке на группу закончится неудачей, поскольку сопоставление с этой группой еще не произошло. Но в последующих попытках ссылка может содержать что-нибудь интересное для вас.

Ниже приводится три типа ссылок на сохраняющие группы. Первый тип – традиционные обратные ссылки, которые будут связаны с реальными данными к моменту обращения к ним:

```
"foofoobar" =~ /^(foo)\1bar$/          # обратная ссылка
```

Следующий тип – опережающие ссылки:

```
"foofoobar" =~ /^((\3bar)|(foo))+$/    # опережающая ссылка
```

К моменту первого обращения по ссылке \3 мы даже не начали сопоставление, поэтому попытка получить подстроку по ссылке \3 потерпит неудачу и будет выполнен переход ко второй альтернативе, которая заполнит третью группу первой строкой "foo". При следующей попытке квантификатора + продолжить сопоставление обращение по ссылке \3 вернет строку "foo", и работа завершится сопоставлением с подстрокой "bar".

В третьем примере ссылка не является ни обратной, ни опережающей, потому что находится внутри группы, на которую ссылается, в результате возникает своего рода циклическая ссылка:

```
"foofoobar" =~ /^(\1bar|(foo))+/       # циклическая ссылка
```

Шаблоны с сохраняющими группами часто применяются в списочном контексте для заполнения списка значениями, поскольку шаблон достаточно «сообразителен», чтобы возвращать захваченные подстроки в виде списка:

```
($first, $last)        = /^(\w+) (\w+)$/;
($full, $first, $last) = /^((\w+) (\w+))$/;
```

При наличии модификатора /g шаблон может вернуть несколько подстрок нескольких найденных соответствий в едином списке. Предположим, что текст почтового заголовка, который мы видели выше, целиком хранится в одной строке (скажем, в $_). То, что делает наш построчный цикл, можно реализовать одной командой:

```
%fields = /^(.*?): (.*)$/gm;
```

Этот шаблон находит четыре соответствия, и в каждом обнаруживает две подстроки. Поиск с модификатором /gm возвращает их все в плоском списке из восьми строк, которые операция присвоения списку %fields удачно интерпретирует как четыре пары ключ/значение, восстанавливая таким образом гармонию во Вселенной.

Есть еще несколько специальных переменных, относящихся к тексту, захваченному при поиске по шаблону. $& содержит всю найденную строку, $` содержит все, что находится слева от соответствия, $' – справа, а $+ содержит текст в последней сохраняющей группе.

```
$_ = "Скажи <EM>друг</EM>, и войдешь.";
m[ (<.*?>) (.*?) (</.*?>) ]x;  # Тег, затем символы, затем закрывающий тег
say "prematch: $`";            # Скажи
say "match: $&";               # <EM>друг</EM>
say "postmatch: $'";           # и войдешь.
say "lastmatch: $+";          # </EM>
```

Описание этих волшебных переменных (и способ их написания по-английски) приводится в главе 25.

Массив @- (@LAST_MATCH_START) **содержит смещения начал всех подчиненных соответствий, а массив** @+ (@LAST_MATCH_END) **содержит смещения концов:**

```
#!/usr/bin/perl
use feature "say";
$alphabet = "abcdefghijklmnopqrstuvwxyz";
$alphabet =~ /(hi).*(stu)/;

say "Соответствие в целом началось в $-[0] и закончились в $+[0]";
say "Первое соответствие началось в $-[1] и закончилось в $+[1]";
say "Второе соответствие началось в $-[2] и закончилось в $+[2]";
```

Если действительно требуется найти буквальный символ скобки, а не интерпретировать его как метасимвол, поставьте перед ним обратную косую черту:

```
/\(т.е. .*?\)/
```

Это соответствует примеру в скобках (т.е. данному уточнению). Но поскольку точка является групповым символом, это соответствует и любому уточнению в скобках с первой буквой т **и третьей буквой** е **(т.е. и этому утверждению).**

Нумерованные сохраняющие группы чрезвычайно чувствительны к изменениям в шаблоне. Представьте, что имеется следующее выражение для поиска повторяющихся слов:

```
$dupword = qr/ \b (?: ( \w+ ) (?: \s+ \1 )+ ) \b /xi;
```

Если встроить его в более крупный шаблон, имеющий свои сохраняющие группы перед этим шаблоном поиска повторяющихся слов, тогда ссылка \1 окажется ошибочной. Например, следующий пример не даст ожидаемого результата:

```
$quoted = qr{ ( ["'] ) $dupword \1 }x
```

потому что теперь, после встраивания в шаблон $quoted, **шаблон в** $dupword ***должен*** **использовать ссылку** \2. **Но это невозможно, потому что на момент компиляции шаблона** $dupword **вторая сохраняющая группа *отсутствует*, и компиляция закончится неудачей.**

Данную конкретную проблему в принципе решают сохраняющие группы с относительной нумерацией. Для доступа к ним следует использовать нотацию \g{*NUMBER*}, **которая указывает на сохраняющую группу с номером** *NUMBER*. **Когда** *NUMBER* **– положительное число, данная форма записи совпадает с формой** *NUMBER*. **Но когда** *NUMBER* **– отрицательное число, оно указывает на предшествующую группу, причем отсчет ведется от текущей позиции в шаблоне в сторону начала. То есть** \g{-1} **– это последняя сохраняющая группа,** \g{-2} **– предпоследняя и т. д.**

Учитывая вышесказанное, шаблон поиска повторяющихся слов, который предполагается встраивать в более крупные шаблоны, лучше будет определить, как показано ниже:

```
$dupword = qr/ \b (?: (\w+) (?: \s+ \g{-1} )+ ) \b /xi;
```

А вот простая программа поиска повторяющихся слов в последовательности абзацев:

```
#!/usr/bin/env perl
use v5.14;

my $dupword = qr{ \b ( \w+ ) (?: \s+ \g{-1} )+ \b }xi;
my $quoted = qr{ ( ["'] ) $dupword \1 }x;
$/ = q();       # через абзацы

while (<>) {
    while (/$quoted/pg) {
        printf "%s %d: %s\n", $ARGV, $., ${^MATCH};
    }
} continue {
    close ARGV if eof;
}
```

Эта программа прекрасно справляется со своими обязанностями, однако проблема никуда не делась. Если встроить $quoted в еще более крупный шаблон, его ссылка \1 также станет неверной. И в нем сложно будет рассчитать, сколько групп следует отступить в обратном направлении, потому что неизвестно, сколько групп имеется во вложенном шаблоне $dupword, никак с ним не связанном.

Именованные сохраняющие группы

Единственный способ решить эту последнюю проблему — использовать иную стратегию, вообще не использующую нумерованные группы. Для этой цели (и многих других) мы изобрели именованные сохраняющие группы. Объявление именованной сохраняющей группы в шаблоне выполняется с помощью конструкции (?<NAME>...). Это обычная сохраняющая группа, которая имеет собственное имя *NAME*.

Или, точнее, она имеет *еще и* имя *NAME*, потому что именованные сохраняющие группы нумеруются так же, как обычные, неименованные сохраняющие группы, и наряду с ними. Таким поведением именованные группы обладают в большинстве библиотек поддержки регулярных выражений для Java и Python. Однако поведение именованных групп несколько отличается в .NET Framework, например, в языке C#, где порядковые номера именованным группам назначаются только после присваивания номеров всем нумерованным группам. (А в другом странном языке, Perl 6, сохраняющая группа получает имя, *только* если она не имеет другого имени. Поди разбери.)

Обратные ссылки на именованные группы в том же самом шаблоне записываются с использованием конструкции \k<*NAME*>. Теперь можно вернуться к определению шаблона $quoted, который доставлял нам неприятности, и подготовить его к интеграции в более крупные шаблоны:

```
$quoted = qr{ (?<quote> ["'] ) $dupword \k<quote> }x;
```

Внутри шаблона все выглядит прекрасно, но, как раньше выполнялось обращение к группе \1 посредством переменной $1 после применения шаблона, вам может потребоваться обратиться к содержимому именованной группы за пределами

приведенного оператора. Этой цели служит встроенный хеш %+. Его ключами являются имена сохраняющих групп в шаблоне, а значениями – фрагменты текста, захваченные этими группами. То есть, принимая во внимание предыдущие определения $quoted и $dupword, извлечь все последовательности повторяющихся слов можно следующим образом:

```
say $+{quote} while /$quoted/g;
```

Вот еще один пример:

```
$word = "bookkeeper";
$word =~ s/ (?<letter> \p{alpha} ) \k<letter> /$+{letter}/gix;
# $word теперь содержит "bokeper"
```

(Модификатор /x использован здесь, только чтобы получить возможность добавить пробелы в шаблон и сделать его более удобочитаемым. Малокомупонравитсячитатьтакойтекст.)

Если так случится, что в одном шаблоне окажется несколько групп с одинаковыми именами, в хеш %+ попадет только первая сохраненная строка, однако имеется другой хеш, %-, хранящий ссылки на массивы сохраненных соответствий. При наличии нескольких групп с одинаковыми именами, для получения содержимого всех групп используйте @{$-{*NAME*}}, где $-{*NAME*}[0] хранит соответствие первой группе, $-{*NAME*}[1] – второй, и так далее, вплоть до последней, $-{*NAME*}[-1].

Представьте, что требуется отыскать имена, за которыми следуют числа, или наоборот – числа, за которыми следуют имена. Если использовать нумерованные сохраняющие группы, возникает проблема – как выяснить, какие группы использовать:

```
/ (\d+) \s+ (\pL+) | (\pL+) \s+ (\d+) /x
```

Поскольку неизвестно, какая из ветвей сработала, нельзя определить, какие группы содержат интересующее нас соответствие – $1 и $2 или $3 и $4. В подобных случаях может пригодиться конструкция сброса ветви (?|...):

```
m{
    (?| (\d+) \s+ (\pL+)       # эти группы доступны как $1 и $2
      | (\pL+) \s+ (\d+)       # и эти группы тоже!
    )
}x
```

В каждой из ветвей перечисления нумерация групп начинается сначала, с числа, предшествовавшего входу в конструкцию выбора ветви. В этой ситуации неважно, с какой половиной будет обнаружено соответствие, поскольку известно, что в данном случае соответствия будут доступны по ссылкам $1 и $2.

Этот способ хорошо подходит для простых шаблонов, но использование именованных сохраняющих групп позволяет справиться с задачей еще лучше, даже в более сложных шаблонах. Если дать группам имена, как показано ниже:

```
m{
    (?<name>  \pL+ ) \s+ (?<number> \d+ )
          |
    (?<number> \d+ ) \s+ (?<name> \pL+ )
}x
```

можно совершенно не волноваться о том, какой вариант в перечислении обнаружит соответствие. Обратиться к группам после сопоставления можно будет, как показано ниже:

```
$+{name}
$+{number}
```

Однако без ветвления обе сохраняющие группы окажутся заполненными дважды:

```
m{
    (?<name>  \pL+ ) \s+ (?<number> \d+ )
    \W+
    (?<number> \d+ ) \s+ (?<name> \pL+ )
}x
```

То есть если найдется соответствие шаблону, на каждую группу <name> и <number>, придется по два соответствия. Когда обнаруживается более одного соответствия для одной именованной группы, предыдущее содержимое переменной %+ не затирается; изменение ее значения происходит только при первом присваивании. Однако переменная %- хранит массивы значений для каждого имени в хеше, поэтому все последующие соответствия будут просто добавляться в конец анонимного массива, связанного с именованным ключом.

Поскольку значениями в %- являются ссылки на массивы, а не строки, как в %+, имеется возможность получить обнаруженные соответствия в виде множества, как показано ниже:

```
@{ $-{name} }
@{ $-{number} }
```

или как отдельные скаляры:

```
$-{name}[0]
$-{name}[1]
$-{number}[0]
$-{number}[1]
```

Отсюда следует, что $+{name} хранит то же значение, что $-{name}[0].

К слову, если вам не по душе переменные, имена которых состоят из единственного знака препинания, можно воспользоваться стандартным модулем Tie::Hash::NamedCapture, который позволяет определять любые имена для этих двух встроенных хешей. Если вы захотите получить версию, которая действует как переменная %-, передайте дополнительный аргумент all => 1, в противном случае вы получите связанный хеш, который действует подобно переменной %+.

```
use Tie::Hash::NamedCapture;
tie my %first_captured, "Tie::Hash::NamedCapture";
tie my %all_captured,  "Tie::Hash::NamedCapture", all => 1;
```

Теперь к соответствиям именованных групп можно обращаться посредством этих переменных, как при использовании %+ и %-, но с другими именами.

```
$first_captured{name}
$first_captured{number}

@{ $all_captured{name} }
@{ $all_captured{number} }
```

```
$all_captured{name}[0]
$all_captured{name}[1]

$all_captured{number}[0]
$all_captured{number}[1]
```

Продемонстрированные приемы использования именованных сохраняющих групп должны были убедить вас в их превосходстве над группами нумерованными в любых шаблонах, кроме самых простых (кто-то скажет, что и в самых простых тоже). Однако по-настоящему все свои преимущества именованные сохраняющие группы начинают раскрывать при использовании в рекурсивных шаблонах и грамматиках, описываемых ниже, в разделе «Замысловатые шаблоны».

Группировка без сохранения

Голые скобки и группируют, и сохраняют. Но иногда это не требуется. Бывает, достаточно просто сгруппировать части шаблона, не создавая ссылки на найденный текст. Можно использовать расширенную форму скобок для подавления сохранения: выражение (?:*PATTERN*) выполняет группировку без сохранения.

Есть по крайней мере три причины, по которым может потребоваться группировка без сохранения:

1. Чтобы квантифицировать части шаблона.
2. Чтобы ограничить область видимости внутреннего перечисления; например, шаблон /^cat|cow|dog$/ должен стать /^(?:cat|cow|dog)$/, чтобы кошка (cat) не убежала с ^.
3. Чтобы ограничить область действия модификатора шаблона конкретным подшаблоном, как, например, в /foo(?-i:Case_Matters)bar/i. (См. следующий раздел, «Замкнутые модификаторы шаблонов».)

Кроме того, отказ от сохранения того, что мы не собираемся использовать, повышает производительность. Отрицательной стороной является несколько более замусоренный внешний вид шаблона.

Применение в шаблоне левой круглой скобки и непосредственно за ней вопросительного знака означает *расширение* регулярного выражения. Бестиарий регулярных выражений в настоящее время относительно фиксирован, и мы не осмеливаемся создать новый метасимвол из опасения нарушить работу старых программ на Perl. Вместо этого для добавления в бестиарий новых функций используется синтаксис расширения.

В оставшейся части этой главы мы увидим много других расширений регулярных выражений, которые группируют без сохранения, а также делают что-то еще. Особенность расширения (?:*PATTERN*) как раз в том, что оно ничего больше не делает. Поэтому высказывание:

```
@fields = split(/\b(?:a|b|c)\b/)
```

похоже на:

```
@fields = split(/\b(a|b|c)\b/)
```

но не возвращает ненужные поля. (Оператор split немного похож на m//g в том, что выдает лишние поля для всех строк, захваченных в шаблоне. Обычно split

возвращает только то, чему соответствий *не* найдено. Дополнительные сведения о split содержатся в главе 27.)

Замкнутые модификаторы шаблонов

Действие модификаторов /i, /m, /s, /x, /d, /u, /a, /l и /p можно ограничить *лексической областью видимости*, поместив их (без обратной косой черты) между символами ? и : в расширенном определении группы. Если сказать:

```
/Harry (?i:s) Truman/
```

то это будет соответствовать обеим строкам, "Harry S Truman" и "Harry s Truman", в то время как:

```
/Harry (?x: [A-Z] \.? \s )?Truman/
```

соответствует "Harry S Truman" и "Harry S. Truman", а также "Harry Truman", а шаблон:

```
/Harry (?ix: [A-Z] \.? \s )?Truman/
```

соответствует всем пяти строкам, объединяя /i и /x внутри группы.

Модификаторы можно «вычитать» из областей видимости с помощью знака «минус»:

```
/Harry (?x-i: [A-Z] \.? \s )?Truman/i
```

Это соответствует любому сочетанию заглавных и строчных букв в имени, но если имя содержит средний инициал, он должен быть заглавной буквой, поскольку действие модификатора /i, примененного к шаблону в целом, отменяется внутри группы.

Выключать таким образом разрешено только модификаторы /i, /m, /s или /x. Остальные модификаторы, /d, /u, /a, /l и /p, можно только включать.

Опустив двоеточие и *PATTERN*, можно экспортировать установки модификаторов в охватывающую группу, превратив ее, таким образом, в область видимости. Это означает, что можно избирательно включать и выключать модификаторы для группы, непосредственно охватывающей данную, например:

```
/(?i)foo/            # Эквивалентно /foo/i
/foo((?-i)bar)/i     # "bar" должно быть в нижнем регистре
/foo((?x-i) bar)/    # Включает /x и отключает /i для "bar"
```

Обратите внимание, что во втором и третьем примерах создаются сохраняющие группы. Если они не нужны, следует применить (?-i:bar) и (?x-i: bar), соответственно.

Ограничение действия модификаторов фрагментом шаблона особенно полезно, когда необходимо, чтобы точка (.) соответствовала символам перевода строки в некоторой части шаблона, но не во всем шаблоне. Установка /s для всего шаблона для этих целей бесполезна.

Перечисление

Внутри шаблона или подшаблона метасимвол | служит для определения набора возможностей, любая из которых считается соответствием. Например:

```
/Gandalf|Saruman|Radagast/
```

соответствует Gandalf или Saruman, или Radagast. Перечисление простирается до ближайшей охватывающей группы (которая может быть как сохраняющей, так и несохраняющей):

```
/prob|n|r|l|ate/       # Соответствует prob, n, r, l или ate
/pro(b|n|r|l)ate/      # Соответствует probate, pronate, prorate или prolate
/pro(?:b|n|r|l)ate/    # Соответствует probate, pronate, prorate или prolate
```

Вторая и третья формы ищут эквивалентные соответствия, но во второй форме переменный символ сохраняется в $1, а в третьей – нет.

В каждой данной позиции Механизм регулярных выражений пытается найти соответствие с первой альтернативой, затем со второй и т. д. Длина альтернатив не имеет значения, поэтому в шаблоне:

```
/(Sam|Samwise)/
```

переменная $1 никогда не получит значения Samwise, в какой бы строке ни производился поиск, поскольку всегда сначала будет найдено соответствие альтернативе Sam. Осуществляя такой поиск с перекрывающимися альтернативами, помещайте более длинные альтернативы в начало шаблона.

Но порядок альтернатив имеет значение только в данной позиции. Внешний цикл Механизма регулярных выражений осуществляет поиск слева направо, поэтому следующее выражение всегда соответствует первой альтернативе Sam:

```
"'Sam I am,' said Samwise" =~ /(Samwise|Sam)/; # $1 eq "Sam"
```

Однако можно предписать просмотр справа налево – при помощи жадных квантификаторов:

```
"'Sam I am,' said Samwise" =~ /.*(Samwise|Sam)/; # $1 eq "Samwise"
```

Можно обойти эффекты направления поиска при просмотре слева направо (или справа налево) посредством различных позиционных утверждений, рассмотренных нами выше, таких как \G, ^ и $. Здесь мы привязываем шаблон к концу строки:

```
"'Sam I am,' said Samwise" =~ /(Samwise|Sam)$/; # $1 eq "Samwise"
```

В этом примере $ выносится за скобки перечисления (поскольку у нас уже есть парочка скобок, после которых можно его поместить), но и в отсутствие скобок можно расставить утверждения перед некоторыми или всеми имеющимися альтернативами соответственно тому, какое соответствие для них нам требуется. В следующей маленькой программе выводятся все строки, начинающиеся с лексем __DATA__ или __END__:

```
#!/usr/bin/perl
while (<>) {
    print if /^__DATA__|^__END__/;
}
```

Но будьте начеку. Помните, что первая и последняя альтернативы (перед первым символом | и вслед за последним) стремятся «поглотить» все элементы регулярного выражения с той или с другой стороны вплоть до конца выражения, если нет охватывающих скобок. Часто встречается такая ошибка:

```
/^cat|dog|cow$/
```

когда в действительности имеется в виду:

```
/^(cat|dog|cow)$/
```

Первый шаблон находит "cat" в начале строки или "dog" в любом месте, или "cow" в конце строки. Второй соответствует любой строке, состоящей только из "cat" или "dog", или "cow". Он также сохраняет результат в переменной $1, что может быть нам не нужно. Избежать сохранения можно одним из двух способов:

```
/^cat$|^dog$|^cow$/
/^(?:cat|dog|cow)$/
```

Альтернатива может быть пустой, и в этом случае она всегда соответствует тексту.

```
/com(pound|)/;      # Соответствует "compound" или "com"
/com(pound(s|)|)/;  # Соответствует "compounds", "compound" или "com"
```

Это похоже на использование квантификатора ?. который находит нуль соответствий или одно соответствие:

```
/com(pound)?/;      # Соответствует "compound" или "com"
/com(pound(s?))?/;  # Соответствует "compounds", "compound" или "com"
/com(pounds?)?/,    # То же, но не использует $2
```

Однако есть одно отличие. Когда ? применяется к шаблону, осуществляющему сохранение соответствия в нумерованной переменной, эта переменная остается неопределенной, если в нее не попадает никакая строка. При использовании пустой альтернативы она тоже ложная, но определена, и в ней находится нулевая строка.

Управление процессом

Всякому хорошему руководителю известно, что не следует чрезмерно опекать своих подчиненных. Нужно просто указать им, что требуется, и позволить самим определять, как лучше всего этого добиться. Аналогично и регулярное выражение зачастую выгоднее всего считать указанием высокого уровня: «Вот что мне нужно; найди-ка мне подходящую строку».

С другой стороны, лучшие руководители также разбираются в той работе, которую пытаются сделать их подчиненные. То же самое верно в отношении поиска по шаблону в Perl. Чем глубже мы понимаем, как Perl решает задачу поиска по некоторому шаблону, тем разумнее мы сможем использовать возможности Perl для поиска по шаблону.

Один из наиболее важных моментов в поиске по шаблону в Perl заключается в том, чтобы понять, когда его применять *не нужно*.

Пусть Perl делает свое дело

У людей с определенным темпераментом знакомство с регулярными выражениями вызывает соблазн любую задачу рассматривать как задачу поиска по шаблону. И хотя это может быть и верно в широком смысле, но поиск по шаблону – это нечто большее, чем простое вычисление регулярных выражений. Отчасти это решение искать ключи от машины там, где вы их уронили, а не под уличным фонарем, где лучше видно. И в повседневной жизни мы все понимаем, что значительно более успешен поиск в правильном месте.

Точно так же следует применять операторы Perl для управления потоком выполнения, чтобы указывать, какие шаблоны выполнять, а какие – пропустить. Регулярные выражения довольно сообразительны, но это сообразительность лошади. Если она видит сразу слишком много, ее внимание может рассеяться. Поэтому иногда нужно надевать на нее шоры. Вспомните приведенный ранее пример перечисления:

```
/Gandalf|Saruman|Radagast/
```

Он работает так, как обещано, но не так хорошо, как мог бы, поскольку Механизм ищет все имена в каждой позиции строки, прежде чем перейти к следующей позиции. Сообразительные читатели «Властелина колец» вспомнят, что из трех названных магов Гэндальф упоминается значительно чаще, чем Саруман, а Саруман упоминается значительно чаще, чем Радагаст. Поэтому обычно для перебора ветвей логические операторы Perl более эффективны:

```
/Гэндальф/ || /Саруман/ || /Радагаст/
```

Это еще один способ преодолеть «левый уклон» Механизма. При этом Сарумана мы ищем, только когда не видно Гэндальфа. А Радагаста – только когда Сарумана тоже нет.

Это позволяет не только изменить порядок поиска объектов, но и (иногда) улучшить работу оптимизатора регулярных выражений. Обычно проще оптимизировать поиск отдельной строки, чем одновременный поиск нескольких строк. Аналогично и не слишком сложный поиск с привязкой тоже часто поддается оптимизации.

Не обязательно ограничивать управление потоком выполнения одним оператором ||. Часто управление возможно на уровне целых операторов. Всегда следует сначала исключать общие случаи. Допустим, мы пишем цикл для обработки файла настройки. Многие файлы с операторами настройки состоят, в основном, из комментариев. Часто лучше всего сразу выбросить комментарии и пустые строки, прежде чем приступать к интенсивной обработке, даже если эта интенсивная обработка выбрасывает комментарии и пустые строки по ходу дела:

```
while (<CONF>) {
    next if /^#/;
    next if /^\s*(#|$)/;
    chomp;
    munchabunch($_);
}
```

Даже если мы не стремимся к повышению производительности, часто требуется чередовать обычные выражения Perl с регулярными выражениями просто потому, что необходимо выполнить какие-то действия, которые невозможно (или трудно) выполнить из регулярного выражения, например осуществить вывод. Вот полезный классификатор чисел:

```
warn "содержит нецифры"      if /\D/;
warn "не натуральное число" unless /^\d+$/;          # отвергает -3
warn "не целое число"       unless /^-?\d+$/;        # отвергает +3
warn "не целое число"       unless /^[+-]?\d+$/;
warn "не десятичное число"  unless /^-?\d+\.?\d*$/; # отвергает .2
warn "не десятичное число"  unless /^-?(?:\d+(?:\.\d*)?|\.\d+)$/;
```

```
warn "не вещественное число языка C"
    unless /^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/;
```

Можно было бы надолго растянуть этот раздел, но, в действительности, такого рода вещам посвящена вся книга. По ходу изложения мы увидим многочисленные примеры взаимодействия кода Perl и поиска по шаблону. В частности, имеется в виду раздел «Программные шаблоны» далее в этой главе. (Конечно, мы не против, если сначала вы прочитаете материал, предшествующий этому разделу.)

Интерполяция переменных

Применение механизмов управления потоком выполнения Perl для управления поиском с использованием регулярных выражений имеет свои ограничения. Основная сложность в том, что это подход типа «все или ничего»; либо мы выполняем шаблон, либо нет. Иногда общие черты требуемого шаблона ясны, но желательно иметь возможность его параметризации. Такую возможность предоставляет интерполяция переменных, подобно тому, как параметризация подпрограммы позволяет в большей мере оказывать влияние на ее выполнение, чем просто решать, вызывать ее или нет. (Подробнее о подпрограммах рассказано в следующей главе.)

Одно из элегантных применений интерполяции состоит в том, чтобы создать некоторую степень абстракции и при этом повысить удобочитаемость. Посредством регулярных выражений можно, несомненно, добиться краткости записи:

```
if ($num =~ /^[-+]?\d+\.?\d*$/) { ... }
```

Но наши намерения становятся более очевидными, если записать так:

```
$sign        = '[-+]?';
$digits      = '\d+';
$decimal     = '\.?';
$more_digits = '\d*';
$number      = "$sign$digits$decimal$more_digits";
...
if ($num =~ /^$number$/o) { ... }
```

Мы еще поговорим о таком использовании интерполяции в разделе «Генерируемые шаблоны» далее в этой главе. Здесь же отметим, что при помощи модификатора /o мы запретили повторную компиляцию шаблона, поскольку предполагаем, что $number не будет изменять свое значение во время выполнения программы.

Другой изящный прием состоит в том, чтобы вывернуть проверки «наизнанку» и задать переменную строку для поиска по шаблону в наборе известных заранее строк:

```
chomp($answer = <STDIN>);
if    ("SEND"  =~ /^\Q$answer/i)  { say "Action is send"  }
elsif ("STOP"  =~ /^\Q$answer/i)  { say "Action is stop"  }
elsif ("ABORT" =~ /^\Q$answer/i)  { say "Action is abort" }
elsif ("LIST"  =~ /^\Q$answer/i)  { say "Action is list"  }
elsif ("EDIT"  =~ /^\Q$answer/i)  { say "Action is edit"  }
```

Этот код позволяет выполнять действие «send», вводя любую строку из S, SE, SEN или SEND (произвольно чередуя верхний и нижний регистры). Для действия «stop» нужно ввести, по крайней мере, ST (или St, или sT, или st).

Когда встречается обратная косая черта

Рассматривая интерполяцию строк в двойных кавычках, мы обычно имеем в виду интерполяцию переменных и обратной косой черты. Но, как уже говорилось, регулярные выражения обрабатываются в два этапа, и на этапе интерполяции интерпретация обратной косой черты, по большей части, откладывается для обработки анализатором регулярных выражений (который мы обсудим ниже). Обычно мы не замечаем разницы, поскольку Perl берет на себя труд ее скрыть.

В действительности очень важно, чтобы анализатор регулярных выражений обрабатывал символ обратной косой черты, потому что только он знает, когда \b обозначает границу слова, а когда символ забоя. Или, допустим, мы выполняем поиск символов табуляции по шаблону с модификатором /x:

```
($col1, $col2) = /(.*?) \t+ (.*?)/x;
```

Если бы Perl не откладывал интерпретацию \t до анализатора регулярных выражений, то \t обратился бы в пробельный символ, который анализатор регулярных выражений и проигнорировал бы из-за модификатора /x. Но Perl не настолько подл или коварен.

Однако есть риск перехитрить самого себя. Предположим, что мы абстрагировали разделитель колонок таким способом:

```
$colsep = "\t+";                        # (двойные кавычки)
($col1, $col2) = /(.*?) $colsep (.*?)/x;
```

Ну вот мы все и испортили, потому что \t превратится в действительный символ табуляции, прежде чем попадет на вход анализатора регулярных выражений, который будет считать, что мы сказали /(.*?)+(.*?)/, после того как выбросит пробельные символы. Чтобы исправить положение, не применяйте /x или используйте одинарные кавычки. А лучше обратитесь к qr//. (См. следующий раздел.)

Единственными экранированными последовательностями в двойных кавычках, обрабатываемыми в этом качестве, являются именованные символы и семь последовательностей трансляции: \N{*CHARNAME*}, \U, \u, \L, \l, \F, \Q и \E. Если бы мы посмотрели, как устроен компилятор регулярных выражений Perl, то нашли бы в нем код для обработки таких экранированных последовательностей, как \t для символа табуляции, \n для символа перевода строки и т. д. Но кода для обработки перечисленных выше экранированных последовательностей мы бы не нашли. (Мы перечислили их в табл. 5.9 только потому, что многие рассчитывают найти их там.) Если каким-то образом нам удастся протащить их в шаблон, не проведя сначала интерполяцию, они не будут распознаны. Если удастся протащить именованный символ, возникнет ошибка, потому что таблица символов, которая была активна при создании строки, должна использоваться и для преобразования имен символов в коды символов. (Это обусловлено тем, что Perl позволяет создавать собственные, нестандартные псевдонимы имен символов, так что не всегда речь идет о стандартном наборе символов. См. раздел «charnames» в главе 29.)

Как могут прокрасться такие символы? Например, через подавление интерполяции – путем использования одиночных кавычек в качестве ограничителя шаблона. В m'...', qr'...' и s'...'...' одинарные кавычки подавляют интерполяцию переменных и обработку экранированных последовательностей, как если бы это были обычные строки, заключенные в одинарные кавычки. Если сказать m'\ufrodo', то бедняга frodo (заглавными буквами) найден не будет. Однако, поскольку «обыч-

ные» символы с обратной косой чертой в любом случае не обрабатываются на этом уровне, m'\t\d' все же соответствует действительному символу табуляции, за которым следует цифра.

Другим способом преодоления интерполяции служит сама интерполяция. Если сказать:

```
$var = '\U';
/${var}frodo/;
```

то несчастный frodo останется в нижнем регистре. Perl не станет повторять проход интерполяции лишь потому, что вы интерполировали нечто, что, похоже, может быть повторно интерполировано. Нельзя рассчитывать, что это будет работать, так же, как и такая двойная интерполяция:

```
$hobbit = "Frodo";
$var = '$hobbit';        # (одинарные кавычки)
/$var/;                  # означает m'$hobbit', а не m'Frodo
```

Вот еще пример, показывающий, что большинство обратных косых черт интерпретируется анализатором регулярных выражений, а не при интерполяции переменных. Представим себе, что есть простая программа в стиле *grep*, написанная на Perl:[1]

```
#!/usr/bin/perl
$pattern = shift;
while (<>) {
    print if /$pattern/;
}
```

Если назвать эту программу *pgrep* и вызвать ее так:

```
% pgrep '\t\d' *.c
```

мы обнаружим, что она выводит все строчки исходного кода на C, в которых за символом табуляции следует цифра. Нам не пришлось делать что-либо особенное, чтобы заставить Perl понять, что \t означает табуляцию. *Будь* шаблоны Perl просто интерполируемыми строками в двойных кавычках, нам бы пришлось; к счастью, они таковыми не являются. Они распознаются непосредственно анализатором регулярных выражений.

Настоящая программа *grep* имеет ключ -i, который отключает поиск с учетом регистра. Этот ключ не нужно добавлять к нашей программе *pgrep*; она и без изменений умеет обрабатывать его. Достаточно передать ей несколько более замысловатый шаблон со встроенным модификатором /i:

```
% pgrep '(?i)ring' LotR*.pod
```

Эта команда найдет «Ring», «ring», «RING» и т. д. Эта функция редко встречается в литеральных шаблонах, поскольку всегда можно просто написать /ring/i. Но для шаблонов, передаваемых в командной строке, в формах поиска в Интернете или встроенных в файлы конфигурации это, может стать настоящим спасательным кругом. (Раз уж зашла речь о кольцах.)

---

[1] Если раньше вы не знали, что за программа *grep*, то теперь узнаете. Ни одна система не обойдется без *grep*: мы считаем *grep* самой полезной программкой из когда-либо созданных. (Отсюда логически следует, что мы не считаем Perl маленькой программой.)

Оператор цитирования регулярных выражений qr/PATTERN/

Переменные, интерполирующиеся в шаблоны, делают это на этапе выполнения, а не компиляции. В былые времена это замедляло выполнение программы, поскольку Perl приходилось проверять, не изменил ли пользователь содержимое переменной; а если изменил, то перекомпилировать регулярное выражение. Сейчас же Perl значительно поумнел, и, чтобы заметить выгоду от практически вымершего модификатора /o (он предписывает Perl интерполировать переменные и компилировать шаблон лишь единожды), придется выполнять интерполяцию в шаблонах, длина которых составляет не менее 10000 символов:

```
print if /$pattern/o;
```

Хотя это прекрасно работает в нашей программе *pgrep*, в общем случае это не работает. Предположим, у нас есть куча шаблонов, и мы собираемся выполнять сопоставление со всеми шаблонами в цикле, возможно, так:

```
for my $item (@data) {
    for my $pat (@patterns) {
        if ($item =~ /$pat/) {    }
    }
}
```

Нельзя написать /$pat/o, поскольку значение $pat изменяется в каждой итерации внутреннего цикла.

Решением является оператор qr/*PATTERN*/msixpodual, который заключает в кавычки и компилирует как регулярное выражение передаваемый ему шаблон *PATTERN*, а кроме того, по понятным причинам, на письме и в речи сокращается до qr//. *PATTERN* интерполируется таким же образом, как в операторе m/*PATTERN*/. Если в качестве ограничителя используется одинарная кавычка ', то интерполяция переменных (или семи экранированных последовательностей трансляции) не производится. Оператор возвращает особое значение, которое можно использовать вместо эквивалентного литерала в соответствующем поиске по шаблону или подстановке. Например,

```
$regex = qr/my.STRING/is;
s/$regex/something else/;
```

эквивалентно:

```
s/my.STRING/something else/is
```

Поэтому в приведенной выше задаче с вложенным циклом обработайте предварительно свои шаблоны в отдельном цикле:

```
@regexes = ();
for my $pat (@patterns) {
    push @regexes, qr/$pat/;
}
```

или разом с помощью оператора Perl map:

```
@regexes = map { qr/$_/ } @patterns;
```

А затем измените цикл, чтобы он использовал прекомпилированные регулярные выражения:

```
for my $item (@data) {
    for my $re (@regexes) {
        if ($item =~ /$re/) { ... }
    }
}
```

Теперь при сопоставлении Perl не придется компилировать регулярное выражение в каждой проверке if, поскольку скомилированное регулярное выражение уже существует.

Результат выполнения qr// можно даже интерполировать в другой шаблон, как если бы этот оператор был простой строкой:

```
$regex = qr/$pattern/;
$string =~ /foo${regex}bar/;   # интерполировать в более крупный шаблон
```

На этот раз Perl перекомпилирует шаблон, но всегда можно связать вместе несколько операторов qr//.

Это работает, потому что оператор qr// возвращает объект особого типа, в котором перегружена операция превращения в строку, как это описано в главе 13. Если вывести возвращаемое значение, то можно увидеть эквивалентную строку:

```
use v5.14;
$re = qr/my.STRING/is;
say $re;                  # выведет (?^usi:my.STRING) в v5.14
```

Символ ^ указывает, что шаблон начинается с установки модификаторов по умолчанию. Модификатор /u указывает, что действует прагма "unicode_strings", поскольку выше сказано: use v5.14. Модификаторы /s и /i включены в шаблоне, потому что мы передали их оператору qr//. Модификаторы же /x и /m не упомянуты, потому что они уже отключены в стандартном окружении, на которое указывает стартовый символ ^. Этот символ сбрасывает установленные пользователем модификаторы и предписывает начать со стандартного набора.

Всякий раз, интерполируя в шаблон строки неизвестного происхождения, следует быть готовым к обработке исключительных ситуаций, возбуждаемых компилятором регулярных выражений, если кто-то подбросит строку с дикими зверюшками:

```
$re = qr/$pat/is;                        # может убежать и съесть вас
$re = eval { qr/$pat/is } || warn ...    # попалась во внешнюю клетку
```

В главе 27 вы узнаете больше об операторе eval.

Компилятор регулярных выражений

После того как этап интерполяции переменных пройден, возможность разобраться с шаблоном, наконец, предоставляется анализатору регулярных выражений. В этот момент неприятности нам практически не грозят, если мы не напутали в скобках и не использовали бессмысленные последовательности метасимволов. Анализатор осуществляет рекурсивно-нисходящий анализ регулярного выражения, и если разбор произведен, оно превращается в формат, пригодный для интерпретации Механизмом (см. следующий раздел). Самое интересное в анализаторе связано с оптимизацией регулярного выражения для максимально быстрого

его выполнения. Эту часть мы излагать не будем. Секрет фирмы. (Слухи о том, что, глядя на код обработки регулярных выражения, можно сойти с ума, сильно преувеличены. Как мы надеемся.)

Зато можно полюбопытствовать, если вам угодно, что анализатор думает о вашем регулярном выражении. Если вежливо его об этом спросить, он ответит. Сказав use re "debug", можно увидеть, как анализатор регулярных выражений обрабатывает шаблон. (Те же сведения можно получить при помощи ключа командной строки -Dr, которым можно пользоваться, если Perl собирали с флагом -DDEBUGGING.)

```
#!/usr/bin/perl
use re "debug";
"Smeagol" =~ /^Sm(.*)[aeiou]l$/;
```

Ниже следует вывод этой программы. Можно видеть, что перед выполнением Perl компилирует регулярное выражение и назначает смысл компонентам шаблона: BOL означает начало строки (^), REG_ANY – точку, и так далее:

```
Compiling REx '^Sm(.*)[aeiou]l$'
Final program:
   1: BOL (2)
   2: EXACT <Sm> (4)
   4: OPEN1 (6)
   6:   STAR (8)
   7:     REG_ANY (0)
   8: CLOSE1 (10)
  10: ANYOF[aeiou][] (21)
  21: EXACT <l> (23)
  23: EOL (24)
  24: END (0)
anchored "Sm" at C floating "l"$ at 3..2147483647 (checking anchored)
anchored(BOL) minlen 4
```

Некоторые строки аннотируют заключения, сделанные оптимизатором регулярных выражений. Ему известно, что строка должна начинаться с букв "Sm", и потому нет оснований для обычного просмотра слева направо. Ему известно, что строка должна оканчиваться буквой "l", поэтому иные строки он без промедления отвергает. Ему известно, что строка должна быть не короче четырех символов, поэтому более короткую строку он игнорирует с лету. Ему также известно, какой символ самый редкий в каждой неизменяющейся строке, что иногда помогает при поиске в «изученных» строках. (См. study в главе 27.)

Затем производится трассировка выполнения сопоставления с шаблоном:

```
Guessing start of match in sv for REx "^Sm(.*)[aeiou]l$" against "Smeagol"
Guessed. match at offset 0
Matching REx "^Sm(.*)[aeiou]l$" against "Smeagol"
0 <> <Smeagol>              |  1:BOL(2)
0 <> <Smeagol>              |  2:EXACT <Sm>(4)
2 <Sm> <eagol>              |  4:OPEN1(6)
2 <Sm> <eagol>              |  6:STAR(8)
                                 REG_ANY can match 5 times
                                 out of 2147483647...
7 <Smeagol> <>              |  8: CLOSE1(10)
```

```
7 <Smeagol> <>              | 10: ANYOF[aeiou][](21)
                                  failed
6 <Smeago> <l>              |  8: CLOSE1(10)
6 <Smeago> <l>              | 10: ANYOF[aeiou][](21)
                                  failed...
5 <Smeag> <ol>              |  8: CLOSE1(10)
5 <Smeag> <ol>              | 10: ANYOF[aeiou][](21)
6 <Smeago> <l>              | 21: EXACT <l>(23)
7 <Smeagol> <>              | 23: EOL(24)
7 <Smeagol> <>              | 24: END(0)
Match successful!
Freeing REx: "^Sm(.*)[aeiou]l$"
```

Если взглянуть на пробелы в слове Smeagol, то можно увидеть, как Механизм регулярных выражений выполняет «перелет», чтобы дать возможность .* проявить максимальную жадность, а затем откатывается назад, пока не станет возможным сопоставить оставшуюся часть шаблона. Но об этом в следующем разделе.

Маленький Механизм, который /(не)? может/

А теперь мы хотим рассказать сказку о Маленьком Механизме Регулярных Выражений, который говорит: «Думаю, что смогу. Думаю, что смогу. Думаю, что смогу».

В этом разделе мы излагаем правила, которыми руководствуется механизм регулярных выражений в Perl для поиска соответствия шаблону в строке. Механизм очень настойчив и трудолюбив. Он часто продолжает работу, когда нам уже кажется, что ему пора сдаться. Механизм не сдастся, пока не будет полностью уверен, что найти соответствие между шаблоном и строкой невозможно. Приведенные ниже Правила объясняют, как Механизм «думает, что сможет» как можно дольше, пока *узнает наверняка*, может он или нет. Сложность задачи для нашего Механизма в том, что она не решается просто грубой силой. Ему приходится вести поиск в потенциально очень сложном пространстве вариантов, отслеживая при этом, что уже проверено, а что еще нет.

Механизм использует для поиска соответствия недетерминированный конечный автомат (NFA, nondeterministic finite-state automaton, который не следует путать с NFL, недетерминированной футбольной лигой). Это означает, что Механизм запоминает, что опробовал, а что – нет, и если что-то не клеится, возвращается и пробует что-то другое. Данный алгоритм известен как *перебор с возвратом* (backtracking); вы уж простите, но этот термин, честно-пречестно, внедрили не мы. Механизм может перепробовать миллион подшаблонов в определенной точке, затем бросить их все, вернуться назад к некоторому варианту в начале и снова перебрать миллион подшаблонов в другой точке. Механизм не слишком умен, просто настойчив и дотошен. Если вы достаточно предусмотрительны, то снабдите Механизму эффективным шаблоном, который не позволит выполнить много ненужных возвратов.

Когда кто-то бросает фразу типа «регулярные выражения выбирают самое левое, самое длинное соответствие», то имеет в виду, что Perl обычно предпочитает самое левое соответствие самому длинному. Но Механизм не осознает, что он что-то «предпочитает», он вообще не думает, а просто потрошит шаблон. Его «предпоч-

тения» возникают как эмерджентное свойство совокупности множества не связанных между собой правил. Вот они:[1]

Правило 1

Механизм пытается найти соответствие в строке как можно левее, чтобы регулярное выражение в целом находило соответствие согласно Правилу 2.

Механизм начинает работу с точки перед первым символом и пытается найти соответствие всему шаблону с этой точки. Весь шаблон соответствует тогда и только тогда, когда Механизм достигает конца шаблона раньше, чем конца строки. Если соответствие найдено, Механизм сразу прекращает работу; продолжать поиск «более удачного» соответствия он не станет, даже если строка может соответствовать шаблону несколькими способами.

Если Механизм не может найти соответствие шаблону, начиная с первой позиции строки, то признает временное поражение и перемещается к следующей позиции в строке, между первым и вторым символами, и снова испытывает все возможности. В случае успеха Механизм останавливается. В случае неудачи продолжает движение по строке. Поиск в целом не считается безуспешным, пока не опробовано все регулярное выражение в каждой позиции строки, в том числе за последним символом.

В строке из *n* символов фактически имеется *n* + 1 позиция для поиска соответствия. Дело в том, что начало и конец соответствия находятся *между* символами строки. Это правило иногда удивляет тех, кто пишет шаблон типа /x*/, который может соответствовать нулю или более символов "x". Если применить этот шаблон к строке "fox", Механизм не найдет "x". Вместо этого он немедленно найдет соответствие нулевой строке перед "f" и больше ничего искать не станет. Чтобы найти один или более символов x, нужно использовать шаблон /x+/. О квантификаторах читайте в Правиле 5.

Следствием этого правила является то, что каждый шаблон, соответствующий пустой строке, будет гарантированно соответствовать самой левой позиции в строке (при отсутствии противоречащих этому утверждений нулевой ширины).

Правило 2

Когда Механизм обнаруживает набор альтернатив (разделенных символами |) – на верхнем уровне, либо на текущем уровне группировки, – он опробует их слева направо, останавливаясь на первом успешном соответствии, позволяющем благополучно завершить поиск шаблона в целом.

Набор альтернатив соответствует строке, если соответствует какая-либо из альтернатив согласно Правилу 3. Если ни одна из альтернатив не соответствует, происходит возврат к Правилу, вызвавшему данное Правило, которым обычно является Правило 1, но это может также быть Правило 4 или 6, если мы находимся внутри группы. Вызывающее правило ищет новую позицию, в которой можно применить Правило 2.

---

[1] Некоторые из этих вариантов могут быть пропущены, если оптимизатор регулярных выражений сказал свое слово, как если бы наш механизм проскочил сквозь гору с помощью квантового туннельного эффекта. Но в данном изложении об оптимизаторе нужно забыть.

Если альтернатива только одна, то либо она соответствует, либо нет, и Правило 2 по-прежнему действует. (Такой вещи, как пустая альтернатива, не существует, поскольку с пустой строкой всегда устанавливается соответствие.)

Правило 3

Любая конкретная альтернатива соответствует строке, если все *элементы*, перечисленные в альтернативе, последовательно находятся в строке согласно Правилам 4 и 5 (так, чтобы при этом удовлетворялось регулярное выражение в целом).

Элемент состоит из *утверждения* (о них говорится в Правиле 4) или *квантифицированного атома* (о них говорится в Правиле 5). Элементам, для которых есть несколько вариантов соответствия, обрабатываются строго слева направо. Если нельзя найти соответствие элементам в заданном порядке, Механизм выполняет возврат и выбирает следующую альтернативу согласно Правилу 2.

Элементы, которые должны быть последовательно найдены, не разделяются в регулярном выражении особыми конструкциями синтаксиса: они просто идут подряд в том порядке, в котором должны оказаться в строке. Когда мы просим найти `/^foo/`, то фактически просим найти четыре элемента, располагающиеся один за другим. Первый является утверждением нулевой ширины, отыскиваемым согласно Правилу 4, а остальные три представляют собой обычные символы, которые должны совпасть сами с собой, один за другим, согласно Правилу 5.

Порядок слева направо означает, что в шаблоне:

```
/x*y*/
```

сначала `x*` выбирает один способ соответствия, а затем `y*` пробует все свои способы. Если последнее завершается неудачей, то `x*` получает возможность выбрать свой второй вариант и заставить `y*` снова перепробовать все свои способы, и т. д. Элементы, расположенные правее, «изменяются быстрее», если позаимствовать выражение у многомерных массивов.

Правило 4

Если утверждение не обнаруживает соответствия в текущей позиции, Механизм возвращается к Правилу 3 и перебирает другие варианты для элементов более высокого приоритета.

Некоторые утверждения более замысловаты, чем другие. В Perl много расширений регулярных выражений, и некоторые из них являются утверждениями нулевой ширины. Например, положительная опережающая проверка `(?=...)` и отрицательная опережающая проверка `(?!...)` не соответствуют каким-либо символам, а просто утверждают, что регулярное выражение `...` должно (или не должно) соответствовать в этой точке, если бы мы гипотетически попробовали его применить.[1]

---

[1] На деле Механизм действительно пытается найти соответствие. Механизм возвращается к Правилу 2, чтобы проверить подшаблон, а затем стирает все упоминания о том, какая часть строки была «проглочена», и возвращает только результат сопоставления – успех или неудачу. (Захваченные подстроки, однако, запоминаются.)

Правило 5

Соответствие атому с квантификатором устанавливается в том случае, когда сам атом соответствует некоторое число раз, допускаемое квантификатором. (Соответствие самого атома устанавливается согласно Правилу 6.)

Для разных квантификаторов требуется разное число соответствий, и большинство квантификаторов допускает использование диапазона совпадений. Многократные соответствия должны идти подряд, т.е. примыкать друг к другу в строке. Для неквантифицированного атома неявно предполагается наличие квантификатора, требующего ровно одного соответствия (иначе говоря, /x/ означает то же самое, что /x{1}/). Если в текущей позиции нельзя найти соответствие допустимому количеству экземпляров атома, Механизм возвращается к Правилу 3 и перебирает другие варианты для элементов более высокого приоритета.

Квантификаторами служат символы и сочетания *, +, ?, *?, +?, ??, *+, ++, ?+, а также различные виды фигурных скобок. Если используется формат {*COUNT*}, то выбора нет, и атом должен соответствовать ровно *COUNT* раз. В противном случае для числа соответствий атома имеется некоторый диапазон, и Механизм отслеживает все выбираемые варианты, чтобы при необходимости осуществить возврат. Но при этом возникает вопрос, с которого из вариантов следует начать. Можно начать с максимального числа и затем снижать его, а можно – с минимального и увеличивать его.

Обычные квантификаторы (без вопросительного знака в конце) предписывают *жадный* поиск; это значит, что они пытаются найти как можно более длинное соответствие. Чтобы найти самое длинное соответствие, Механизму приходится проявлять некоторую осторожность. Неверное предположение может обойтись очень дорого, поэтому на практике Механизм не производит обратный отсчет от самого большого значения, которое может быть Очень Большим и повлечь миллионы неправильных предположений. Поэтому Механизм поступает несколько умнее: сначала он *подсчитывает*, сколько соответствующих атомов (подряд) в действительности имеется в строке, а затем использует *этот* фактический максимум в качестве первого выбранного значения. (Он также запоминает все более короткие варианты на случай, когда самый длинный не вписывается.) Затем он (наконец-то) пытается найти соответствие для оставшейся части шаблона, предполагая, что самый длинный вариант является самым лучшим. Если для самого длинного варианта оказывается невозможно найти соответствие с оставшейся частью шаблона, происходит возврат и опробуется следующий по длине вариант.

Если, например, мы говорим /.*foo/, то Механизм пытается найти максимальное число «любых» символов (представленных точкой) вплоть до конца строки, прежде чем пытаться найти "foo"; и когда для "foo" там не находится соответствия (и не может найтись, потому что в конце строки для него недостаточно места), Механизм начинает «сдавать» по одному символу с конца, пока не найдет "foo". Если "foo" встречается в строке более одного раза, механизм остановится на последнем вхождении, поскольку в действительности это будет *первое* найденное при поиске с возвратом соответствие. Когда найден весь шаблон при некоторой определенной длине .*, Механизм знает, что может теперь

отбросить все более короткие варианты для `.*` (те, которые она стала бы использовать, если бы текущее "foo" не подошло).

Завершая жадный квантификатор вопросительным знаком, мы превращаем его в умеренный квантификатор, который для первой попытки выбирает самое маленькое количество соответствий. Поэтому, если сказать `/.*?foo/`, то `.*?` сначала пытается найти соответствие 0 символов, затем 1 символу, затем 2 и так далее, пока не будет найдено соответствие "foo". Вместо того чтобы отступать назад, Механизм, так сказать, отступает вперед и заканчивает работу, найдя в строке первое вхождение "foo", а не последнее.

Правило 6

Поиск каждого атома производится на основе семантики его типа. Если соответствие атому не найдено (или найденное не позволяет успешно сопоставить оставшуюся часть шаблона), Механизм откатывается к Правилу 5 и опробует другой вариант количества экземпляров атома.

Соответствие атомов определяется согласно следующим типам.

- Регулярное выражение в круглых скобках, `(...)`, соответствует тому же, чему соответствует регулярное выражение (представленное `...`) согласно Правилу 2. Скобки действуют, как оператор группировки для квантифицирования. Скобки имеют также побочный эффект, сохраняя найденную подстроку, которую можно потом использовать в *ссылке на группу* (часто называемой *обратной ссылкой*). Побочный эффект можно подавить расширением конструкции `(?:...)`, которая имеет только группирующую семантику: она ничего не запоминает в `$1`, `$2` и других переменных. Возможны и другие формы атомов (и утверждений) в скобках – см. оставшийся материал этой главы.
- Точка соответствует любому символу, за исключением, возможно, символа перевода строки.
- Список символов в квадратных скобках (*класс символов*) соответствует любому из символов, указанных в списке.
- Буква, предваренная обратной косой чертой, соответствует конкретному символу или символу из числа приведенных в табл. 5.9.
- Любой другой символ, предваренный обратной косой чертой, соответствует этому символу.
- Любой символ, не упомянутый выше, соответствует самому себе.

Все это звучит довольно сложно, но ведем мы к следующему. Для каждого набора вариантов, созданного квантификатором или перечислением, у Механизма регулярных выражений есть ручка, которую можно покрутить. Он будет крутить эти ручки, пока не найдется соответствие шаблону в целом. Правила просто говорят, в каком порядке механизму регулярных выражений позволено крутить эти ручки. Слова «механизм регулярных выражений предпочитает самое левое соответствие» просто означают, что ручку начальной позиции он крутит медленнее всего. А поиск с возвратом означает, что нужно открутить назад ручку, которую мы только что крутили, и попытаться покрутить ручку, находящуюся выше в порядке старшинства, положение которой изменяется медленнее.

Вот более конкретный пример – программа, которая для пары смежных слов обнаруживает соответствие окончания одного слова началу другого:

```
$a = 'nobody',
$b = 'bodysnatcher';
if ("$a $b" =~ /^(\w+)(\w+) \2(\w+)$/) {
    print "$2 перекрывается в $1-$2-$3\n"
}
```

Программа выведет:

```
body перекрывается в no-body-snatcher
```

Можно предположить, что $1 из-за своей жадности сначала захватит все слово "nobody". Так и происходит – поначалу. Но когда это сделано, не остается символов для переменной $2, которые необходимо поместить в нее из-за квантификатора +. Поэтому механизм регулярных выражений отступает, и $1 неохотно отдает один символ для $2. На этот раз успешно отыскивается пробел, но затем механизм видит \2, которая представляет жалкую букву y. Следующий символ в строке не y, а b. Механизму приходится постоянно отступать и делать еще попытки, и, в конечном итоге, $1 отдает слово «body» переменной $2. Habeas corpus,[1] так сказать.

На самом деле, это не вполне срабатывает, когда само перекрытие является удвоением, как, например, в паре слов "rococo" и "cocoon". Приведенный выше алгоритм решит, что перекрывающейся строкой, $2, будет "co", а не "coco". Но нам не нужно "rocococoon"; нам нужно "rococoon". Это тот случай, когда вы можете оказаться умнее Механизма. Добавив квантификатор минимального соответствия к части $1, получим превосходный шаблон /^(\w+?)(\w+) \2(\w+)$/, который делает как раз то, что нужно.

Значительно более подробное обсуждение достоинств и недостатков различных типов механизмов регулярных выражений можно найти в книге Джеффри Фридла (Jeffrey Friedl) «Mastering regular expressions».[2] Механизм регулярных выражений Perl очень хорошо выполняет многие повседневные задачи, которые требуется выполнять с помощью Perl, и успешно решает даже не вполне повседневные, если проявить к нему чуточку уважения и понимания.

Замысловатые шаблоны

Опережающие и ретроспективные проверки

Иногда требуется только глянуть украдкой. Существует четыре расширения регулярных выражений, которые позволяют сделать это и которые мы называем *проверками обстановки* (*lookaround assertions*), поскольку они позволяют выполнить своего рода разведку, не осуществляя настоящий поиск символов. Эти проверки позволяют выявить соответствие некоторому шаблону, которое было бы (или не было бы) найдено, если бы мы попытались это сделать. Механизм решает

[1] Акт «Habeas Corpus», гарантирующий свободу личности, принятый в Англии в 1679 г., согласно которому судебная власть обязана по требованию заинтересованных лиц обеспечить доставку в суд всякого, кто подвергся заключению, для проверки законности ареста. – *Прим. ред.*

[2] Джеффри Фридл «Регулярные выражения», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008.

задачу, осуществляя действительный поиск соответствия гипотетическому шаблону и делая потом вид, что оно не найдено (если найдено).

Когда Механизм заглядывает вперед из текущей позиции строки, мы называем это *опережающей проверкой* (*lookahead assertion*). Если он оглядывается назад – *ретроспективной проверкой* (*lookbehind assertion*). Шаблоны для опережающей проверки могут быть любыми регулярными выражениями, но шаблоны для ретроспективной проверки могут иметь только фиксированную ширину, поскольку Механизму требуется знать, откуда начинать сопоставление.

Хотя все эти четыре расширения являются утверждениями нулевой ширины и потому не поглощают символы (по крайней мере, официально), они позволяют захватывать строки – достаточно добавить дополнительные уровни сохраняющих скобок.

(?=*PATTERN*) *(положительная опережающая проверка)*

Когда Механизм встречает конструкцию(?=*PATTERN*), он заглядывает дальше в строку и проверяет *наличие* соответствия шаблону *PATTERN*. Если вы помните, в нашей программе для удаления повторяющихся слов пришлось написать цикл, поскольку шаблон каждый раз съедал слишком много:

```
$_ = "Paris in THE THE THE THE spring.";

# удалить слова, повторяющиеся дважды (и трижды (и четырежды...))
1 while s/\b(\w+) \1\b/$1/gi;
```

Услышав слова «съел слишком много», всегда нужно подумать об опережающей проверке. (Ну, почти всегда.) Заглядывая вперед, вместо того, чтобы проглотить второе слово, можно написать программу, удаляющую дубликаты за один проход, например:

```
s/ \b(\w+) \s (?= \1\b ) //gxi;
```

Конечно, это не совсем правильный код, поскольку он может исказить и корректные фразы вроде «The clothes you DON DON't fit».

Опережающие проверки можно использовать для реализации перекрывающихся соответствий. Например,

```
"0123456789" =~ /(\d{3})/g
```

вернет всего три строки: 012, 345 и 678. Обернув сохраняющую группу опережающей проверкой:

```
"0123456789" =~ /(?=(\d{3}))/g
```

можно получить все перекрывающиеся соответствия: 012, 123, 234, 345, 456, 567, 678 и 789. Действие данного фрагмента кода основано на том, что утверждение заглядывает вперед, чтобы отыскать и сохранить соответствие, но, будучи опережающей проверкой, не поглощает ни одного символа. Когда Механизм обнаруживает, что должен повторить попытку из-за использования модификатора /g, то перешагивает через один символ.

(?!*PATTERN*) *(отрицательная опережающая проверка)*

Когда Механизм встречает конструкцию (?!*PATTERN*), он заглядывает дальше в строку и проверяет *отсутствие* соответствия шаблону *PATTERN*. Чтобы исправить предшествующий пример, можно добавить отрицательную опережающую

проверку вслед за положительной опережающей проверкой и исключить случай сокращенной формы:

```
s/ \b(\w+) \s (?= \1\b (?! '\w))//xgi;
```

Метасимвол \w в конце необходим, чтобы можно было различать сокращение и слово в конце предложения в одинарных кавычках. Можно пойти еще дальше, чтобы законные обороты, такие как «that that particular», наша программа не «исправляла». Мы добавим альтернативу для отрицательной опережающей проверки, чтобы предотвратить исправление «that» (и таким образом покажем, что для группировки альтернатив может использоваться любая пара круглых скобок):

```
s/ \b(\w+) \s (?= \1\b (?! '\w | \s particular))//gix;
```

Теперь мы знаем, что фраза «that that particular» в безопасности. К несчастью, Геттисбергская речь[1] по-прежнему под угрозой. Поэтому добавим еще одно исключение:

```
s/ \b(\w+) \s (?= \1\b (?! '\w | \s particular | \s nation))//igx;
```

Это уже начинает выходить из-под контроля. Поэтому составим Официальный Список Исключений и прибегнем к хитрому трюку с интерполяцией, чтобы переменная $" разделяла альтернативы символом |:

```
@thatthat = qw(particular nation);
local $" = '|';
s/ \b(\w+) \s (?= \1\b (?! '\w | \s (?: @thatthat )))//xig;
```

(?<=PATTERN) (***положительная ретроспективная проверка***)

Когда Механизм обнаруживает конструкцию (?<=PATTERN), он «оглядывается» в строке и проверяет наличие соответствия шаблону PATTERN.

В нашем примере все еще остается проблема. Хотя он уже позволяет Честному Эйбу произносить такие фразы, как «that that nation», он также допускает фразы типа «Paris, in the the nation of France». Мы можем добавить положительную ретроспективную проверку перед нашим списком исключений, чтобы гарантировать применение исключений @thatthat только к «that that»:

```
s/ \b(\w+) \s (?= \1\b (?! '\w | (?<= that) \s (?: @thatthat )))//ixg;
```

Да, это становится ужасно сложным, но потому этот раздел и назван «Замысловатые шаблоны», в конце концов. Если вам потребуется сделать шаблон еще более замысловатым, чем к данному моменту сделали мы, разумное применение комментариев и qr// поможет вам сохранить здравомыслие. Или, по крайней мере, сохранить большую его часть.

Или подумайте об использовании метасимвола \K, чтобы обмануть Механизм относительно того, где официально начинается соответствие. В этом случае предыдущий шаблон будет действовать как своего рода ретроспективная проверка для официальной части шаблона, но сканирование текста будет выполняться слева направо. Это особенно удобно, когда возникает потребность изменять

---

[1] Речь Авраама Линкольна, произнесенная в 1863 году и считающаяся образцом ораторского искусства. Содержит оборот «that that nation» – «чтобы эта нация». – *Прим. перев.*

глубину ретроспективной проверки, чего невозможно добиться другими средствами Механизма.

(?<!*PATTERN*) ***(отрицательная ретроспективная проверка)***

Обнаружив конструкцию (?<!=*PATTERN*), механизм «оглядывается» назад в строке и проверяет *отсутствие* соответствия шаблону *PATTERN*.

На этот раз возьмем совсем простой пример. Простой вариант старого орфографического правила: «I перед E, если только не после C». На Perl ошибки исправляются так:

```
s/(?<!c)ei/ie/g
```

Программист должен сам решить для себя, будет ли он обрабатывать какие-либо исключения. (Например, «weird» пишется как «weird», а произносится как «wierd».)

Неуступающие группы

Как описывалось в разделе «Маленький Механизм, который /(не)? может/», при сопоставлении с шаблоном механизм регулярных выражений часто осуществляет перебор с возвратом. Можно помешать механизму регулярных выражений осуществлять возврат к некоторому набору вариантов при помощи *неуступающей группировки*. Неуступающая группировка (?>*PATTERN*) работает так же, как простая группировка (?:*PATTERN*), но, если она обнаруживает соответствие шаблону *PATTERN*, возврат к другим квантификаторам или альтернативам внутри подшаблона подавляется. (Следовательно, бессмысленно использовать такой подход с шаблоном, не содержащим квантификаторов или альтернатив.) Единственный способ заставить Механизм изменить свое решение – это выполнить возврат к чему-либо до подшаблона в невозвращающей группе и повторно войти в подшаблон слева.

Это похоже на посещение автодилера. Всласть наторговавшись, вы ставите ультиматум: «Вот мое последнее предложение; либо оно принимается, либо мы расходимся». Если оно не принимается, вы не начинаете снова торговаться. Вместо этого вы возвращаетесь восвояси – через входную дверь. Можно пойти к другому дилеру и опять начать торговаться. Вам позволено снова торговаться, но только потому, что вы снова вошли в неуступающую группу в другом контексте.

Приверженцы языков Prolog и SNOBOL могут представить это себе как ограничивающий возвраты оператор отсечения (scoped cut) и, соответственно, оператор «забора» (fence operator).

Посмотрите, как в `"aaab" =~ /(?:a*)ab/` подшаблон `a*` сначала находит три `a`, а потом отдает одну из них, поскольку последняя буква `a` понадобится позже. Подгруппа жертвует частью того, что ей требуется, чтобы успешным оказалось сопоставление в целом. (Продолжая нашу аналогию – вы разрешаете продавцу автомобиля уговорить вас на бóльшую цену, поскольку боитесь, что иначе сделка не состоится вовсе.) Напротив, подшаблон в `"aaab" =~ /(?>a*)ab/` никогда не отдаст того, что захватил, даже если такое его поведение приведет к невозможности сопоставления в целом. (Как поется в песне, ты должен понимать, когда отвечать, когда пасовать, а когда из-за стола вставать.)

Хотя группировка (?>*PATTERN*) полезна для изменения поведения шаблона, она применяется в основном, чтобы ускорить получение неудачного результата при

некоторых видах поиска, когда известно, что он все равно окажется неудачным (если только не будет успешным сразу). Механизму может потребоваться на удивление много времени, чтобы осознать бесперспективность поиска, особенно когда речь о вложенных квантификаторах. Следующий шаблон почти мгновенно приводит к успеху:

```
$_ = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab";
/a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*[Bb]/;
```

Но проблема не в успехе. Проблема в неудаче. Если удалить из строки последнюю букву b, то сопоставление с шаблоном, возможно, будет выполняться долгие и долгие годы, прежде чем будет завершиться с отрицательным результатом. Многие и многие тысячелетия. Фактически миллиарды и миллиарды лет.[1] Внимательно посмотрев, мы обнаружим, что этот шаблон не может быть успешно сопоставлен, если в конце строки нет буквы b, однако оптимизатор регулярных выражений не настолько сообразителен (на момент написания книги), чтобы определить, что /[Bb]/ никогда не найдется. Но если дать ему намек, то можно заставить его быстро определить неудачу, в то же время позволяя получить успех, когда это возможно:

```
/(?>a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*)[Bb]/;
```

Возьмем более реалистичный, как мы надеемся, пример. Представим себе программу, которая считывает текст по абзацу за раз и показывает строки, имеющие продолжение – строки, отмеченные обратной косой чертой в конце строки. Вот образец из *Makefile* для Perl, в котором действует данное соглашение по продолжению строк:

```
# Files to be built with variable substitution before miniperl
# is available.
sh = Makefile.SH cflags.SH config_h.SH makeaperl.SH makedepend.SH \
        makedir.SH myconfig.SH writemain.SH
```

Можно написать такую простую программу:

```
#!/usr/bin/perl -00p
use feature "say";
while ( /( (.+) ( (?<=\\) \n .* )+ ) /gx) {
    say "GOT $.: $1\n";
}
```

Она работает правильно, но очень медленно. Дело в том, что Механизм отступает от конца строки с шагом в один символ, сокращая то, что находится в $1. Это неумно. Если избавиться от ненужных захватов, это не сильно поможет. Шаблон:

```
(.+(?:(?<=\\)\n.*)+)
```

несколько ускоряет работу, но незначительно. В такой ситуации существенно помогает неуступающая группировка. Шаблон:

```
((?>.+)(?:(?<=\\)\n.*)+)
```

---

[1] В реальности речь, скорее, о многих септиллионах лет. Мы не можем сказать точно, сколько времени потребуется. Мы не стали дожидаться завершения работы в надежде на положительный результат. В любом случае, компьютер откажет раньше, чем случится тепловая смерть Вселенной, а это регулярное выражение закончит свою работу позже, чем произойдет любое из этих событий.

делает то же самое, но на порядок быстрее, поскольку не тратит время на возврат в поисках того, чего нет.

С помощью (?>...) нельзя добиться такого успеха, которого нельзя добиться с помощью (?:...) и даже простого (...). Но если провал неизбежен, лучше пусть он произойдет скорее, чтобы можно было жить дальше.

Между прочим, поскольку наш пример содержит единственный квантификатор, группировку (?>.+) можно было бы записать более кратко, как .++.

Программные шаблоны

Большинство программ на Perl следует императивному (называемому также процедурным) стилю программирования, в котором набор отдельных команд подается в понятном порядке: «Разогреть плиту, перемешать, глазировать, нагреть, остудить, подать пришельцам.» Иногда в эту смесь добавляют чуть-чуть функционального программирования («Возьмите немного больше глазури, чем, как вам кажется, нужно, даже после прочтения этого пункта инструкции, рекурсивно») или сдабривают ее объектно-ориентированной технологией («но, пожалуйста, придержите объекты анчоусов»). Часто встречается комбинация всех этих стилей.

Но Механизм регулярных выражений применяет для решения задач совершенно иной подход, являющийся скорее декларативным. Мы описываем цели на языке регулярных выражений, а Механизм реализует ту логику, которая требуется для решения наших задач. Языки логического программирования (например, Prolog) не всегда освещаются так же хорошо, как языки, где доминируют остальные три стиля, но используются чаще, чем это может показаться. Perl нельзя было бы даже скомпилировать без *make*(1) или *yacc*(1), которые можно рассматривать если не как чисто декларативные языки, то, по крайней мере, как гибриды, в которых соединились императивное и логическое программирование.

В Perl тоже можно поступать подобным образом, смешивая декларации целей с императивным кодом более смело, чем мы это делали до сих пор, и опираясь на сильные стороны обоих подходов. Можно программным образом создавать строку, которая, в конечном счете, попадет на вход Механизма, в известном смысле создавая программу, которая пишет новую программу «на лету».

Обычные выражения на Perl можно также передавать в качестве заменяющей части оператора s/// с помощью модификатора /e. Это позволяет динамически генерировать строку замены путем выполнения фрагмента кода при обнаружении каждого соответствия шаблону.

Еще более изощренным приемом является внедрение в любые точки шаблона фрагментов кода с помощью расширения (?{*CODE*}). Такой фрагмент будет выполняться каждый раз, когда Механизм сталкивается с ним, перемещаясь по шаблону вперед и назад в сложном танце сопоставления с возвратами.

Наконец, можно применить s///ee или (??{*CODE*}) для введения еще одного уровня косвенности: сами результаты выполнения этих фрагментов кода будут повторно вычисляться и использоваться, создавая фрагменты программы и шаблона на лету в нужный момент.

Генерируемые шаблоны

Программы, пишущие другие программы, – самые счастливые программы на свете.»[1] В книге Джеффри Фридла «Mastering regular expressions» окончательная демонстрация силы заключается в написании программы, создающей регулярное выражение для определения того, удовлетворяет ли строка требованиям RFC 822, т.е. содержится ли в строке совместимый со стандартами допустимый почтовый заголовок. Шаблон на выходе этой программы имеет длину в несколько тысяч символов, и читать его примерно так же легко, как двоичный дамп памяти, создаваемый при аварийном завершении программы. Однако Механизм Perl такими вещами не смутить; компиляция шаблона происходит «без сучка, без задоринки» и, что еще более интересно, поиск выполняется очень быстро – значительно быстрее, чем для многих коротких шаблонов, требующих сложного поиска с возвратами.

Это очень сложный пример. А выше мы привели очень простой пример того же подхода, когда построили шаблон `$number` из деталей (см. раздел «Интерполяция переменных»). Но чтобы продемонстрировать мощь этого программного подхода при создании шаблона, возьмем задачу средней сложности.

Допустим, что необходимо выбрать все слова с определенной последовательностью гласных и согласных; например, «audio» и «eerie» удовлетворяют шаблону VVCVV (ГГСГГ)[2]. Хотя нетрудно описать, что считать гласной, а что согласной, вряд ли вам захочется каждый раз заново набирать это на клавиатуре. Даже в нашем простом случае (VVCVV) потребуется набрать шаблон, который выглядит примерно так:

```
^[aeiouy][aeiouy][cbdfghjklmnpqrstvwxzy][aeiouy][aeiouy]$
```

Более универсальная программа должна получить строку VVCVV и программно создать такой шаблон. Еще лучше, если она сможет принять слово вроде audio, использовать его как макет для получения строки VVCVV, а из нее уже получить требуемый шаблон. Кажется сложным, но на практике это не так, поскольку шаблон будет генерировать программа, такая как программа *cvmap* ниже:

```
#!/usr/bin/perl
use v5.14;
use feature "say";
use warnings;
my $vowels = 'aeiouy';
my $cons   = 'cbdfghjklmnpqrstvwxzy';
my %map = (C => $cons, V => $vowels); # исходное отображение для C (С) и V (Г)

for my $class ($vowels, $cons) {        # теперь для каждого типа букв
    for (split //, $class) {           # получить все буквы этого типа
        $map{$_}  = $class;            # и отобразить букву обратно в тип
    }
}

my $pat = "^";
for my $char (split //, shift) {       # для каждой буквы в слове шаблона
    $pat .= "[$map{$char}]";           # добавить соответствующий класс символа
```

---

1 Эндрю Юмом (Andrew Hume), известный философ UNIX.

2 V – первая буква слова «vowel» (гласная), а C – слова «consonant» (согласная). – *Прим. ред.*

```
}

my $re = qr/^${pat}$/i;                  # компилировать шаблон
say "REGEX is $re";                      # отладочная выдача
@ARGV = ('/usr/share/dict/words')        # выбрать словарь по умолчанию
    if -t && !@ARGV;

while (<>) {                             # пройти через входные данные,
    print if /$re/;                      # выводя все соответствующие строки
}
```

Все интересное содержится в переменной `%map`**. Ключами этого хеша служат буквы алфавита, а соответствующими значениями – все буквы их типа. C и V тоже есть в ней, поэтому можно задать либо** `VVCVV`**, либо** `audio`**, и в любом случае получить** `eerie`**. Каждый символ аргумента, переданного программе, используется для извлечения нужного класса символов и добавления его в шаблон. Когда шаблон создан и скомпилирован с помощью** `qr//`**, сопоставление (даже для длинного шаблона) происходит быстро. Вот что можно получить, если запустить эту программу с аргументом** `fortuitously` **(случайно, неожиданно):**

```
% cvmap fortuitously /usr/dict/words
REGEX is (?^ui:^[cbdfghjklmnpqrstvwxzy][aeiouy][cbdfghjklmnpqrstvw
xzy][cbdfghjklmnpqrstvwxzy][aeiouy][aeiouy][cbdfghjklmnpqrstvwxzy][a
eiouy][aeiouy][cbdfghjklmnpqrstvwxzy][cbdfghjklmnpqrstvwxzy][aeiouyc
bdfghjklmnpqrstvwxzy]$)
carriageable
circuitously
fortuitously
languorously
marriageable
milquetoasts
sesquiquarta
sesquiquinta
villainously
```

Глядя на это регулярное выражение, можно убедиться в том, какого объема гнусного ввода с клавиатуры удалось избежать благодаря «ленивому» программированию, хотя и окольным путем.

Вычисления при подстановке

Когда модификатор `/e` **(«е» от «expression evaluation» – вычисление выражения) используется в выражении** `s/PATTERN/CODE/e`**, заменяющая часть интерпретируется не как строка в двойных кавычках, а как выражение на языке Perl. Это похоже на встроенную команду** `do { CODE }`**. Несмотря на внешний вид строки это, в действительности, блок кода, компилируемый одновременно с остальной программой – задолго до того, как фактически будет производиться подстановка.**

Модификатор `/e` **позволяет создавать строки замены более изощренным способом, чем интерполяция в двойных кавычках. Разницу можно показать на примере:**

```
s/(\d+)/$1 * 2/;      # Заменяет "42" на "42 * 2"
s/(\d+)/$1 * 2/e;     # Заменяет "42" на "84"
```

А вот как температура по Цельсию переводится в температуру по шкале Фаренгейта:

```
$_ = "Разогреть духовку до 233C.\n";
s/\b(\d+\.?\d*)C\b/int($1 * 1.8 + 32) . "F"/e; # преобразование в 451F
```

Этому приему можно придумать бесконечное число применений. Вот фильтр, который модифицирует поступающие в него файлы по месту (как редактор), добавляя 100 к каждому числу, которым начинается строчка (и за которым следует двоеточие, на которое мы только смотрим, но не ищем или заменяем):

```
% perl -pi -e 's/^(\d+)(?=:)/100 + $1/e' filename
```

Иногда нужно сделать несколько больше, чем просто использовать найденную строку в другом вычислении. Иногда нужно, чтобы строка и *была* вычислением, результат которого выступит в качестве заменяющего значения. Каждый дополнительный модификатор /e вслед за первым заключает код, который нужно выполнить, в eval. Следующие две строки эквивалентны, но первую легче прочесть:

```
s/PATTERN/CODE/ee
s/PATTERN/eval(CODE)/e
```

Этот прием подходит для замены упоминаний простых скалярных переменных на их значения:

```
s/(\$\w+)/$1/eeg;        # Интерполяция большинства значений скаляров
```

Поскольку в действительности используется eval, модификатор /ee находит даже лексические переменные. Вот этот, слегка более развитой пример вычисляет замену для простых арифметических выражений с (неотрицательными) целыми числами:

```
$_ = "I have 4 + 19 dollars and 8/2 cents.\n"
s{ (
        \d+ \s*          # найти целое число
        [+*/-]           # и арифметический оператор
        \s* \d+          # и второе целое число
   )
}{ $1 }eegx;             # затем разыменовать $1 и выполнить этот код
print;                   # "I have 23 dollars and 4 cents.
```

Как и для любых других вхождений eval *STRING*, перехватываются ошибки этапа компиляции (например, синтаксические) и исключительные ситуации этапа выполнения (например, деление на ноль). Если они возникают, то узнать, что произошло, можно из переменной $@ ($EVAL_ERROR).

Выполнение кода на этапе сопоставления

В большинстве программ, использующих регулярные выражения, логической последовательностью выполняемых команд руководит управляющая структура этапа выполнения самой программы. Мы пишем условные инструкции if или циклы while, либо вызываем функции или методы, которые могут привести к вызову операции сопоставления с шаблоном. Даже в s///e управление принадлежит оператору подстановки, а код, находящийся в строке замены, выполняется только при успехе поиска.

В случае применения *подшаблонов, содержащих код*, обычная связь между регулярным выражением и программным кодом инвертируется. Когда Механизм применяет к нашему шаблону свои Правила на этапе поиска, он может наткнуть-

ся на расширение регулярного выражения вида (?{*CODE*}). При срабатывании этот подшаблон не выполняет сопоставление или «заглядывание». Это утверждение нулевой ширины, которое всегда «успешно» и выполняется только для получения побочных эффектов. Когда в процессе сопоставления с шаблоном Механизму нужно пройти подшаблон, содержащий код, он выполняет этот код.

```
"glyph" =~ /.+ (?{ say "hi" }) ./x; # Выведет "hi" два раза.
```

Когда Механизм пытается сопоставить строку glyph с этим шаблоном, он сначала позволяет .+ «съесть» все пять букв. Затем выводит hi. Когда Механизм находит последнюю точку, все пять букв уже «съедены», поэтому он отступает к .+ и заставляет его отдать одну из пяти букв. Затем он снова двигается вперед по шаблону, останавливаясь, чтобы еще раз вывести hi, перед тем как присвоить h последней точке и успешно завершить поиск.

Фигурные скобки вокруг фрагмента *CODE* напоминают, что это — блок кода на языке Perl, и он определенно ведет себя как блок в лексическом смысле. Это означает, что если в таком блоке для объявления переменной с лексической областью видимости используется my, то переменная является частной для этого блока. Но если локализовать переменную с динамической областью видимости при помощи local, результат может оказаться не таким, как мы ожидаем. Подшаблон (?{*CODE*}) создает неявную динамическую область видимости, которая действует в оставшейся части шаблона, пока сопоставление не завершится успехом, либо произойдет возврат через подшаблон с кодом. Можно представлять себе это так, будто возврат из блока по достижении его конца не происходит. Вместо этого осуществляется невидимый рекурсивный вызов Механизма для поиска соответствия оставшейся части шаблона. Лишь по завершении этого рекурсивного вызова происходит возврат из блока и снятие локализации с переменных.[1] В следующем примере мы инициализируем переменную $i значением 0, включив в начало шаблона подшаблон, содержащий код. Затем мы находим соответствие любому числу символов с помощью .* — но между . и * мы поместим еще один подшаблон с кодом, чтобы подсчитать, сколько раз . находит соответствие.

```
$_ = 'lothlorien',
m/ (?{ $i = 0 })                # Установить $i в 0
   (.   (?{ $i++ }) )*          # Обновить $i, даже после возврата
   lori                         # Вызывает возврат
 /x;
```

Наш Механизм летит вперед, устанавливая $i в 0 и позволяя .* проглотить все 10 символов строки. Встретив в шаблоне литерал lori, он отступает и отдает эти

[1] Тех, кто знаком с нисходящими рекурсивными анализаторами, такое поведение может смутить, поскольку подобные компиляторы возвращаются из вызова рекурсивной функции, как только что-то вычислят. Механизм же этого не делает: когда он что-то вычислит, то уходит глубже в рекурсию (даже выходя из группы в круглых скобках!). Нисходящий рекурсивный анализатор находится в положении минимальной рекурсии, когда в конце успешно завершает разбор, но наш Механизм находится в локальном *максимуме* рекурсии, когда в конце шаблона успешно завершает сопоставления. Может оказаться полезным мысленно подвесить шаблон за левый край и представить себе его как дерево графа вызовов. Если вы можете создать эту картину у себя в голове, то динамическая область видимости локальных переменных становится более понятной. (Если не можете, то хуже вам не станет.)

четыре символа из `.*`. После того, как найдено соответствие, `$i` по-прежнему будет равна 10.

Чтобы в `$i` находилось число символов, которые в итоге окажутся в `.*`, можно использовать в шаблоне динамическую область видимости:

```
$_ = 'lothlorien';
m/ (?{ $i = 0 })
   (. (?{ local $i = $i + 1; }) )* # Обновление $i,не зависящее от возвратов.
   lori
   (?{ $result = $i })            # Копировать по нелокализованному адресу.
/x;
```

Здесь посредством `local` мы гарантируем хранение в `$i` числа символов, найденных `.*`, независимо от наличия возвратов при поиске. `$i` будет забыта, когда закончится регулярное выражение, поэтому подшаблон с кодом `(?{ $result = $i })` сохраняет этот счетчик в `$result`.

Специальная переменная `$^R` (описанная в главе 25) содержит результат последнего блока `(?{CODE})`, выполненного в составе успешного сопоставления.

Расширение `(?{CODE})` можно использовать в качестве условия *COND* в конструкции `(?(COND) IFTRUE|IFFALSE)`. При этом `$^R` не устанавливается, а круглые скобки вокруг условного оператора можно опустить:

```
"glyph" =~ /.+(?(?{ $foo{bar} gt "symbol" }) .|signet)./;
```

Здесь мы проверяем, больше ли `$foo{bar}`, чем `symbol`. Если да, то в шаблон включается `.`, а если нет, то `signet`. Несколько растянув эту команду, можно сделать ее более удобочитаемой:

```
"glyph" =~ m{
    .+                          # несколько любых
    (?(?{                       # если
            $foo{bar} gt "symbol" # это правда
        })
            .                   # найти еще любой символ
        |                       # иначе
            signet              # найти signet
    )
    .                           # и еще любой символ
}x;
```

Когда действует прагма `use re 'eval'`, регулярное выражение может содержать подшаблоны `(?{CODE})`, даже если в них интерполируются переменные:

```
/(.*?) (?{length($1) < 3 && warn}) $suffix/;   # Ошибка без use re "eval"
```

Обычно это запрещено, поскольку представляет собой угрозу безопасности. Даже если приведенный шаблон невинен, поскольку невинен `$suffix`, анализатор регулярных выражений не может различить, какие части строки интерполировались, а какие нет, поэтому он просто запрещает все подшаблоны с кодом, если производилась интерполяция.

Если шаблон получен из «меченых» данных, даже `use re "eval"` не разрешит поиск по шаблону.

Когда действует прагма use re "taint", и к меченой строке применяется регулярное выражение, сохраненные соответствия подшаблонам (в нумерованных переменных или списке значений, возвращаемом m// в списочном контексте) помечаются. Это удобно, если операции регулярного выражения над мечеными данными имеют целью не извлечение безопасных подстрок, а просто осуществление других преобразований. Дополнительные сведения о меченых данных см. в главе 20. Для этой прагмы заранее скомпилированные регулярные выражения (обычно получаемые через qr//) не считаются интерполированными:

```
/foo${pat}bar/
```

Это допустимо, если $pat представляет собой скомпилированное регулярное выражение, даже если $pat содержит подшаблоны с кодом (?{*CODE*}).

Выше мы привели небольшой фрагмент вывода use re 'debug'. Более простое решение для отладки заключается в использовании подшаблонов с кодом (?{*CODE*}) для вывода соответствия, уже найденного в процессе поиска:

```
"abcdef" =~ / .+ (?{say "Пока найдено: $&"}) .cdef $/x;
```

В результате выводится:

```
Пока найдено: abcdef
Пока найдено: abcde
Пока найдено: abcd
Пока найдено: abc
Пока найдено: ab
Пока найдено: a
```

Здесь видно, как .+ захватил все буквы, и отдает их по одной – по мере того, как Механизм выполняет возвраты.

Интерполяция шаблона на этапе сопоставления

Фрагменты шаблона можно создавать прямо в шаблоне. Расширение (??{*CODE*}) делает возможным добавление кода, результатом выполнения которого является допустимый шаблон. Похоже на выражение вроде $pattern, за исключением того, что $pattern можно генерировать на этапе выполнения или, точнее, на этапе сопоставления. Например:

```
/\w (??{ if ($threshold > 1) { "red" } else { "blue" } }) \d/x;
```

Это эквивалентно /\wred\d/, если $threshold больше 1, и /\wblue\d/ в противном случае.

Выполняемый код может содержать ссылки на группы только что найденных подстрок (даже если позже они окажутся «ненайденными» из-за возврата). Например, следующая команда находит все строки, которые одинаково читаются в прямом и обратном направлении (известные как палиндромадеры, фразы «с горбом посередине»):

```
/^ (.+) .? (??{quotemeta reverse $1}) $/xi;
```

Сбалансировать круглые скобки можно так:

```
$text =~ /( \(+ ) (.*?) (??{ '\)' x length $1 })/x;
```

Это соответствует строкам вида (shazam!) и (((shazam!))), помещая shazam! в $2. К несчастью, этот шаблон не определяет, сбалансированы ли вложенные скобки. Для этого требуется рекурсия.

По счастливому стечению обстоятельств, существует возможность создавать и рекурсивные шаблоны. Можно заставить скомпилированный шаблон, использующий (??{*CODE*}), ссылаться на себя самого. Рекурсивный поиск оказывается довольно нерегулярным. В любой книге по регулярным выражениям указывается, что стандартные регулярные выражения не способны правильно находить вложенные скобки. И это правильно. Правильно и то, что регулярные выражения в Perl не являются стандартными. Следующий шаблон[1] находит набор вложенных скобок на любом уровне вложенности:

```
$np = qr{
           \(
           (?:
              (?> [^()]+ )    # Нескобки без возврата
            |
              (??{ $np })     # Группировать с найденными скобками
           )*
           \)
        }x;
```

Вот как можно с его помощью организовать поиск вызова функции:

```
$funpat = qr/\w+$np/;
"myfunfun(1,(2*(3+4)),5)" =~ /^$funpat$/;      # Соответствует!
```

Условная интерполяция

Расширение регулярных выражений (?(*COND*)*IFTRUE*|*IFFALSE*) аналогично оператору ?: в Perl. Если условие *COND* истинно, применяется шаблон *IFTRUE*; в противном случае – шаблон *IFFALSE*. Условие *COND* может быть ссылкой на найденный текст (выражаемой как «голое» целое число без \ или $), опережающей или ретроспективной проверкой, или подшаблоном с кодом. (См. разделы «Опережающие и ретроспективные проверки» и «Выполнение кода на этапе сопоставления» ранее в этой главе.)

Если условие *COND* представляет собой целое число, оно считается ссылкой на сохраняющую группу. Рассмотрим пример:

```
#!/usr/bin/perl
use feature "say";
$x = 'Perl является бесплатным.';
$y = 'ManagerWare стоит $99.95.';

foreach ($x, $y) {
    /^(\w+) (?:является|(стоит)) (?(2)(\$\d+)|\w+)/; # Либо (\$\d+), либо \w+
    if ($3) {
        say "$1 стоит денег"                   # ManagerWare стоит денег
    } else {
        say "$1 распространяется бесплатно"; # Perl распространяется бесплатно
    }
}
```

1 Обратите внимание, что нельзя объявить переменную в той же команде, в которой мы собираемся ее использовать. Разумеется, всегда можно объявить ее заранее.

Здесь *COND* есть (2), что истинно, когда существует ссылка на второй найденный текст. Если это так, то (\$\d+) включается в шаблон в этой точке (создавая ссылку на найденный текст $3); в противном случае используется \w+.

Если условие *COND* представляет собой опережающую или ретроспективную проверку, или подшаблон, содержащий код, то определить, надо ли включать IFTRUE или IFFALSE, можно на основании истинности утверждения:

```
/[ATGC]+(?(?<=AA)G|C)$/;
```

Здесь в качестве *COND* используется ретроспективная проверка, чтобы найти в ДНК последовательность, которая оканчивается на AAG или другую основную комбинацию и C.

Альтернативу |*IFFALSE* можно опустить. В этом случае шаблон *IFTRUE* будет включен в шаблон, как обычно, если истинно *COND*, но если условие не истинно, Механизм перейдет к следующей части шаблона.

Рекурсивные шаблоны

Ссылаясь на сохраняющую группу из шаблона, вы добавляете фрагмент, фактически сохраненный группой, на которую указывает обратная ссылка.

```
/\b (        \p{alpha}+ ) \s+    \1     \b /x # нумерованная  обратная ссылка
/\b (        \p{alpha}+ ) \s+  \g{1}    \b /x # альтернативный синтаксис
/\b (        \p{alpha}+ ) \s+ \g{-1}    \b /x # относительная обратная ссылка
/\b (?<word> \p{alpha}+ ) \s+ \k<word> \b /x # именованная   обратная ссылка
```

Во всех этих четырех примерах обратные ссылки используются, чтобы отыскать в строке повторяющиеся слова. Но иногда бывает желательно с помощью одного и того же шаблона отыскать два разных слова.

Для этого используется другая конструкция: (?*NUMBER*) снова вызывает шаблон, заключенный в нумерованную группу, а (?&*NAME*) – заключенный в именованную группу. (Последняя конструкция напоминает форму &-вызова подпрограммы.)

```
/\b (        \p{alpha}+ ) \s+ (?-1)     \b /x # "вызвать" нумерованную группу
/\b (?<word> \p{alpha}+ ) \s+ (?&word) \b /x # "вызвать" именованную группу
```

При использовании формы *NUMBER* ведущий минус указывает, что отсчет групп должен производиться справа налево, от текущей позиции, т.е. -1 означает вызов предыдущей группы, абсолютную позицию которой в этом случае знать совсем не обязательно. С другой стороны, если перед номером стоит знак плюс, (?+*NUMBER*), отсчет выполняется в противоположном направлении, слева направо, от текущей позиции.

Можно даже вызвать группу из нее самой, вынудив механизм сопоставления выполнить рекурсивный вызов. Это вполне нормальное явление. Ниже демонстрируется один из вариантов реализации поиска парных скобок:

```
/ ( \( (?: [^()]++ | (?1) )*+ \) )/x
```

Поскольку весь шаблон заключен в сохраняющие скобки, их можно опустить и использовать (?0) для вызова «нулевой группы», под которой подразумевается весь шаблон целиком:

```
/ \( (?: [^()]++ | (?0) )*+ \) /x
```

В данном случае шаблон прекрасно работает, но его работа может нарушиться, если определить его с помощью оператора qr// и задействовать в другом шаблоне. В последнем случае следует использовать относительную нумерацию групп. Ниже приводится регулярное выражение, отыскивающее идентификатор, за которым следуют парные скобки:

```
$funcall = qr/\w+ ( \( (?: [^()]++ | (?-1) )*+ \) )/x
```

Использовать его следует таким образом:

```
while (<>) {
    say $1 if /^ \s* ( $funcall ) \s* ; \s* $/x;
}
```

В переменной $1 остается только желаемый результат, что удобно. Это пример кода, не зависящего от абсолютной позиции: он не требует знания абсолютной позиции в общей схеме.

Нумерованные группы прекрасно работают в простых шаблонах, но с ростом сложности шаблона именованные группы становятся более привлекательной альтернативой:

```
$funcall = qr/\w+ (?<paren> \( (?: [^()]++ | (?&paren) )*+ \) )/x
while (<>) {
    say $+{func} if /^ \s* (?<func> $funcall ) \s* ; \s* $/x,
}
```

По существу, именованные группы – единственный способ сохранить здравомыслие с ростом сложности шаблонов.

Обратите внимание, что вложенный вызов не возвращает сохраненный им фрагмент в охватывающий шаблон – он возвращает позицию найденного соответствия. Зачастую это именно то, что необходимо, потому что позволяет контролировать побочные эффекты, но иногда бывает желательно сохранить фрагменты анализируемой строки. Подробнее об этом чуть ниже.

Раз уж речь зашла о разборе текста: как вы, быть может, осознали к этому моменту, у вас на руках есть практически все необходимое для настоящего анализа. Под настоящим анализом мы не имеем в виду простые конечные автоматы, такие как NFA и DFA, а боевой, рекурсивно-нисходящий разбор. Благодаря особенностям Perl, описываемым в следующем разделе и далее, шаблоны превращаются в полноценный эквивалент рекурсивно-нисходящего анализатора. Добавив пару рюшечек и бантиков, вы без труда сможете писать полноценные грамматики, весьма напоминающие файлы *yacc* или грамматики в форме Бэкуса-Наура (Backus-Naur Form).

Грамматические шаблоны

Действие грамматик основано на рекурсивном анализе подшаблонов. Поскольку сохраняющие группы могут работать как подшаблоны, их можно использовать в качестве порождающих правил грамматик. До настоящего моменты вы еще не видели, как определять порождающие правила отдельно от их вызова.

Очевидно, если использовать сохраняющую группу как подшаблон, не имеет значения, применяется ли она фактически для сопоставления с чем-либо. В действительности, когда пишутся грамматики, обычно не предполагается, что подшабло-

ны будут вызываться сразу же после их определения – вы сначала определяете их все, а используете позднее. Что нужно, так это способ указать, что определенный фрагмент шаблона служит исключительно для определения, а не для выполнения.

Именно это делают блоки (?(DEFINE)...). Технически – это условная инструкция, подобная конструкции (?(*COND*)...), описанной выше. Здесь условием является DEFINE, и это условие всегда ложное.[1] Поэтому содержимое блока DEFINE гарантированно не будет выполняться.

Теперь самоуверенный всезнайка из аудитории может заявить, что компилятор волен удалять неиспользуемый код из тех условных конструкций, где условное выражение всегда возвращает ложное значение. Впрочем, преподаватель может парировать, что эта политика действует только для исполняемого кода – не для объявлений. А группы в блоке DEFINE считаются именно объявлениями, поэтому компилятор их не выбрасывает. Они служат подшаблонами, доступными для вызова в другом месте того же шаблона.

Таким образом, вот то место, куда можно поместить нашу грамматику. Блок DEFINE можно определить в любом месте, где только пожелаете, но обычно его располагают либо непосредственно перед основным шаблоном, либо вслед за ним. Порядок следования определений внутри блока не имеет значения. Поэтому все ваши подшаблоны наверняка будут напоминать следующее:

```
qr{
    (?&abstract_description_of_what_is_being_matched)
    (?(DEFINE)
        (?<abstract_description_of_what_is_being_matched>
            (?&component1)
            (?&component2)
            (?&component3)
            ...
        )
        (?<component1> ... )
        (?<component2> ... )
        (?<component3> ... )
        ...
    )
}x;
```

Единственный исполняемый фрагмент этого шаблона располагается за пределами блока DEFINE: он вызывает первое правило, которое вызывает все остальные.

Это начинает походить не только на определение грамматики, но и на обычную программу. В отличие от привычной формы обработки шаблона слева направо, данное определение имеет форму программы, выполняемой сверху вниз и состоящей из подпрограмм, вызывающих друг друга итеративно и рекурсивно. Невозможно преувеличить важность этой модели разработки, потому что выбор понятных имен является единственно важным подходом, упрощающим создание, отладку и сопровождение шаблонов. В этом разработка шаблонов ничем не отличается от обычного программирования.

1 Да, это «колхоз», но вполне удачный. В противном случае пришлось бы использовать постфиксный квантификатор {0}, как в языке Ruby, в ущерб удобочитаемости.

И данная модель позволяет сразу заметить отсутствие необходимых абстракций – если вы начинаете повторяться, повторяющуюся функциональность, вероятно, имеет смысл оформить как отдельный именованный элемент. А если вы дадите ему говорящее имя, позволяющее определить, для чего этот элемент служит, это упростит организацию и сопровождение кода. Если позднее потребуется внести какие-то изменения, достаточно будет сделать это всего в одном месте. Подпрограммы были самой первой парадигмой повторного использования кода, а подшаблоны – это те же подпрограммы, только замаскированные.

Такой стиль определения шаблонов быстро вызывает привыкание, стоит только попробовать: нетривиальные шаблоны перестают выглядеть как обычные регулярные выражения и приобретают черты своих мощных собратьев, грамматик. Вам больше не придется писать:

```
/\b(?=(?=[\p{Alphabetic}\p{Digit}])\X)(?:(?=[\p{Alphabetic}\p{Digit}])\X
|['\x{2019}]|(?=[^\x{2014}])\p{dash})+(?!(?=[\p{Alphabetic}\p{Digit}])\X|)/
```

Вместо этого вы сможете использовать *предпочтительный* способ определения грамматического шаблона для извлечения слов:

```
$word_rx = qr{ (?&one_word)
    (?(DEFINE)
        (?<a_letter>     (?= [\p{Alphabetic}\p{Digit}] ) \X          )
        (?<some_letters> (?: (?&a_letter) | (?&tick) | (?&dash) ) + )
        (?<tick>         ['\N{RIGHT SINGLE QUOTATION MARK}]          )
        (?<dash>         (?= [^\N{EM DASH}] ) \p{dash}               )
        (?<one_word>
            \b
            (?= (?&a_letter) )
            (?&some_letters)
            (?! (?&a_letter)
              | (?&dash)
            )
        )
    ) # конец блока определения
}x;
```

Шаблон верхнего уровня просто вызывает подпрограмму one_word, которая и выполняет всю работу. Этот шаблон позволяет вывести все слова в файл, по одному в строке:

```
While (/($word_rx)/) {
      say $1;
}
```

Как демонстрируют эти примеры, иногда грамматические шаблоны можно записывать, как обычные, неграмматические регулярные выражения, не имеющие подпрограмм-шаблонов. Но это крайне неудобно и такие выражения, где все свалено в кучу, невероятно сложны для чтения, написания, отладки и сопровождения. Грамматики расширяют ваши возможности. Многие интереснейшие задачи невозможно решить с применением обычных регулярных выражений. Рассмотрим пример задачи, легко решаемой с применением шаблонов Perl, но недоступной для решения с помощью обычных регулярных выражений. (Точнее, два таких примера, один связан с поиском сбалансированных (balanced) тегов, а другой – зеркальных (mirrored).)

Сама эта книга написана в формате POD, используемом в системе создания документации Perl. Подробнее о нем рассказывается в главе 23, а здесь отметим лишь, что формат POD немного напоминает любой другой язык разметки на основе SGML. Он имеет теги и угловые скобки. И как в любом другом языке разметки, эти теги могут быть вложенными, что делает бессмысленными попытки поиска или манипулирования ими с помощью старых добрых инструментов, основанных на регулярных выражениях, таких как почтенный *grep*. Теги образуют вложенные структуры, так что операции поиска и манипуляции тегами также должны иметь многоуровневую организацию. А простые регулярные выражения этого делать не позволяют. К счастью, регулярные выражения в Perl – это не обычные регулярные выражения, поэтому они могут использоваться для анализа весьма замысловатых документов на таких языках разметки, как XML и HTML. И POD.

Теги в языке разметки POD всегда начинаются с одиночной буквы в верхнем регистре, за которой следует одна или более открывающих угловых скобок. Пока все просто. Вся сложность состоит в том, чтобы отыскать закрывающую угловую скобку (или скобки) в конце тега. И сделать это можно двумя способами, в зависимости от наличия пробельного символа за открывающей комбинацией.

```
X<содержимое>        # сбалансированный тег
X<< содержимое >>    # зеркальный тег
```

Если пробельный символ отсутствует, значит вы имеете дело со сбалансированным (balanced) тегом, где количество закрывающих скобок должно соответствовать числу открывающих, включая любые скобки в «начинке».

При наличии пробела тег превращается в зеркальный (mirrored): он завершается пробельным символом, за которым следует число закрывающих скобок, соответствующее числу открывающих. Внутри тега может присутствовать любое количество открывающих или закрывающих угловых скобок, и они н обязаны быть парными, так как их число никак не учитывается.

Ниже приводится несколько примеров тегов из текста книги:

```
B<-0xR<HHH...>>
C<<< >>= >>>
C<0..(2Y<R<BITS>>)-1>
C<< BZ<><touch> SZ<><BZ<><-t> IZ<><time>> IZ<><file> >>
C<(?E<lt>!...)>
C<< !grep { !R<CODE>->($_) } keys R<HASH> >>
C<<< <HANDLE>, <<END >>>
C<(?(R<COND>)R<IFTRUE>|R<IFFALSE>)>
C<<< << R<EXPR> >>>
C<s/R<PATTERN>/R<REPLACEMENT>/>
I<Santa MarE<iacute>a>
X<<< < (left angle bracket);<< (left-shift) operator:@leftleft >>>
```

А вот начало определения соответствующей грамматики:

```
podtag::  capital either
capital:: uppercase_letter
either::  < balanced | mirrored >
```

Она легко транслируется непосредственно в подшаблоны:

```
(?<podtag>    (?&capital) (?&either)      )
```

```
(?<capital>    \p{uppercase_letter}        )
(?<either>     (?&balanced) | (?&mirrored) )
```

Группа сбалансированных угловых скобок – это просто текст, заключенный в парные угловые скобки, такой как B<-0xR<HHHH>>. Мы уже знаем, как отыскивать фрагменты, заключенные в парные угловые скобки, потому что эта задача ничем не отличается от задачи поиска парных круглых скобок, которая была решена выше.

```
(?<balanced> < ( [^<>]++ | (?&balanced) )* > )
```

Это подводит нас к последней части нашей грамматики, зеркальным тегам. Зеркальные теги – это теги, которые выглядят, как C<<< << R<EXPR> >>>. В этом случае требуется отыскать закрывающие скобки, число которых соответствует числу открывающих скобок, при этом не должны учитываться открывающие и закрывающие угловые скобки, находящиеся внутри тега. Ну, почти.

```
(?<mirrored> (?<open> <{2,}+ ) \s++
                \s+
                  (?: (?&podtag) | \p{Any} ) *?
                \s+
             \s++ (??{ ">" x length $MATCH{open} })
)
```

Сначала группа <mirrored> безвозвратно поглощает две или более открывающих скобки и сохраняет их в группе <open>, чтобы позднее можно было подсчитать их количество. Обратите внимание, что здесь именованная группа <open> не выделяется в отдельное именованное правило, так как применение вложенного правила скрыло бы найденное соответствие.

Средняя часть соответствует содержимому тега. Здесь нам следует проявлять осторожность, потому что это содержимое может включать другие теги POD, а *эти* теги могут быть сбалансированными. Все, что не является зеркальным тегом, не учитывается. Здесь используется свойство \p{Any}, которое на языке Юникода соответствует выражению (?s:.) на языке Perl. Иными словами, ему соответствуют любые символы, даже символы перевода строки.[1]

Последняя строка интерполирует результат этого выражения, чтобы сгенерировать последовательность закрывающих угловых скобок, обнаруженных в строке 1.

Ниже приводится программа целиком.

```
#!/usr/bin/env perl
# demo-podtags-core
use v5.14;
use strict;
use warnings;
use open qw(:std :utf8);       # весь текст в кодировке UTF-8
use warnings FATAL => "utf8";  # на случай появления ошибок с кодировкой
use re "/x";                   # все шаблоны допускают форматирование
our %MATCH; *MATCH = \%+,      # %MATCH - псевдоним переменной %+ для удобства

my $RX = qr{
```

[1] Технически это не так. На более высоких уровнях соответствия стандартам, чем чем обеспечивает Perl, \p{Any} может совпадать с такими специфическими национальными символами, как комбинации из двух или трех букв.

```
    (?(DEFINE)
        (?<podtag>      (?&capital) (?&either)           )
        (?<capital>     \p{upper}                        )
        (?<either>      (?&mirrored) | (?&balanced)      )
        (?<balanced>    < (?&contents) >                 )
        (?<contents>    (?: (?&podtag) | (?&unangle) )* )
        (?<unangle>     [^<>]+                           )
        (?<mirrored>    (?<open> <{2,}+ ) \s++
                           \s+
                               (?: (?&podtag) | \p{Any} ) *?
                           \s+
                        \s++ (??{ ">" x length $MATCH{open} })
        )
    )
};

@ARGV = glob("*.pod")        if @ARGV == 0 && -t STDIN;
die "usage: $0 [pods]\n"     if @ARGV == 0 && -t STDIN;

$/ = "";        # режим деления на абзацы, поскольку теги могут
                # пересекать \n, но не \n\n
$| = 1;         # быстрый вывод для нетерпеливых

while (<>) {
    while (/ (?<TAG> (?&podtag) ) $RX /g) {
        say $MATCH{TAG};
    }
}
```

Пара замечаний. Чтобы вывести результат сопоставления, его необходимо явно сохранить где-то в области видимости. Для этого используется именованная группа <TAG>. Можно было бы использовать $& или ${^MATCH}, однако все, что сохраняется в группах *внутри* шаблона <podtag>, теряется после возврата из него.

Этот прием можно использовать для проверки входных данных, но когда дело доходит до извлечения фрагментов, поиск которых дался таким трудом, начинают проявляться недостатки. Можно усеять код вставками (?{ CODE }), сохраняющими найденные фрагменты, но это быстро превращается в весьма утомительное занятие. Более удачное решение заключается в использовании вспомогательного модуля, который описывается в следующем разделе.

Модуль Grammars

Дамиан Конвей (Damian Conway) написал модуль Regexp::Grammars (его можно найти в архиве CPAN), решающий эту проблему и многие другие. Он упрощает создание грамматик на Perl. Модуль просто великолепен, но слишком многогранный, чтобы описать его здесь, поэтому подробности ищите на соответствующих страницах справочного руководства. И все же вот кое-что, чтобы пробудить у вас интерес.

Модуль Regexp::Grammars в действительности является компилятором грамматик, напоминающим *yacc*. Отличие от *yacc*: вместо программы на языке C Regexp::Grammars генерирует шаблон, который можно использовать как любой другой. Он делает это за счет одной хитрости, о которой будет рассказываться в следующем разделе: он перегружает оператор qr// и переписывает шаблон, который вы ему даете, в другой, который делает всю грязную работу.

Ниже приводится версия предыдущей программы, использующая модуль Дамиана.

```
#!/usr/bin/env perl
# demo-podtags-grammar
use v5.14;
use strict;
use warnings;
use open qw(:std :utf8);      # весь текст в кодировке UTF-8
use warnings FATAL => "utf8"; # на случай появления ошибок с кодировкой
use re "/x";                  # все шаблоны допускают форматирование

my $podtag = do { use Regexp::Grammars; qr{
    <podtag>
    <token: podtag>        <capital> <either>
    <token: capital>       \p{upper}
    <token: either>        <mirrored> | <balanced>
    <token: balanced>      \< <contents> \>
    <token: contents>      (?: <[podtag]> | <[unangle]> ) *
    <token: unangle>       [^<>]+
    <token: mirrored>      <open=( \< {2,} )>
                               \s+
                                   (?: <podtag> | \p{Any} ) *?
                               \s+
                           </open>
    }xms;
};

@ARGV = glob("*.pod")          if @ARGV == 0 && -t STDIN;
die "usage: $0 [pods]\n"       if @ARGV == 0 && -t STDIN;

$/ = "";        # режим деления на абзацы, поскольку теги могут пересекать \n, но не \n\n
$| = 1;         # быстрый вывод для нетерпеливых

while (<>) {
    while (/$podtag/g) {
        say $/{podtag}{capital},
            $/{podtag}{either}{""}
    }
}
```

Эта программа выполняет анализ тех же входных данных, что и предыдущая – определение грамматики выглядит почти так же. Но она лучше во многих отношениях. Ее проще было написать, и на наш взгляд она проще читается. Кроме того, она делает кое-что, чего не делает предыдущая версия. Взгляните на подпрограмму <mirrored>. **Здесь используется одна из возможностей модуля** Regexp::Grammars, **позволяющая сохранять открывающие угловые скобки в группе с именем** <open> **и затем неявно сопоставлять количество закрывающих угловых скобок, просто сказав** </open>.

Самое, пожалуй, важное заключается в том, что и шаблон, выполняющий сопоставление, и инструкции вывода немного изменились. Сопоставление теперь выглядит намного проще, а вот вывод усложнился. Переменная-хеш с именем %/ хранит вложенную структуру данных, созданную в результате успешного поиска, выполненного грамматическим регулярным выражением.

В этом листинге не видно, что переменная %/ имеет сложную структуру. Она в точности соответствует результатам анализа. Допустим, что на вход программы был передан текст: «Left C<B<nested>> an I<< N<inside>un>tag >> gets X<indexed> right». После его анализа мы воспользуемся модулем Data::Dump, чтобы показать вам, что находится в переменной %/.

```
use Data::Dump; # из CPAN
# выполните сопоставление и затем
dd \%/;          # передайте ссылку на хеш с результатами
```

И вы получите:

```
{
  "" => "C<B<nested>>",
  "podtag" => {
    "" => "C<B<nested>>",
    "capital" => "C",
    "either" => {
      "" => "<B<nested>>",
      "balanced" => {
        "" => "<B<nested>>",
        "contents" => {
          "" => "B<nested>",
          "podtag" => [
            {
              "" => "B<nested>",
              "capital" => "B",
              "either" => {
                "" => "<nested>",
                "balanced" => {
                  "" => "<nested>",
                  "contents" => { "" => "nested", "unangle" => ["nested"] }
                },
              },
            },
          ],
        },
      },
    },
  },
}
{
  "" => "I<< N<inside>un>tag >>",
  "podtag" => {
    "" => "I<< N<inside>un>tag >>",
    "capital" => "I",
    "either" => {
      "" => "<< N<inside>un>tag >>",
      "mirrored" => {
        "" => "<< N<inside>un>tag >>",
        "open" => "<<",
        "podtag" => {
          "" => "N<inside>",
          "capital" => "N",
          "either" => {
```

```
              "" => "<inside>",
              "balanced" => {
                "" => "<inside>",
                "contents" => { "" => "inside", "unangle" => ["inside"] },
              },
            },
          },
        },
      },
    },
  }
  {
    "" => "X<indexed>"
    "podtag" => {
      "" => "X<indexed>"
      "capital" => "X",
      "either" => {
        "" => "<indexed>",
        "balanced" => {
          "" => "<indexed>",
          "contents" => { "" => "indexed", "unangle" => ["indexed"] },
        },
      },
    },
  }
```

Как видите, внутри хеша хранятся все результаты применения правил грамматики, полученные в ходе анализа, включая вложенные теги. Модуль Regexp::Grammars обладает намного более широкими возможностями, чем было показано здесь. Плюс ко всему, если вам доведется столкнуться с созданием своих грамматик, он позволит даже разным грамматикам совместно использовать подпрограммы регулярных выражений друг друга. Возможности совместного использования намного шире, чем в простых грамматиках, представленных в предыдущем разделе, где нет ни одного совместно используемого элемента. Вам стоит всерьез изучить этот модуль, если вы занимаетесь реализацией сложного синтаксического анализа.

Определение собственных утверждений

Нельзя изменить (легко: см. следующий раздел) принцип действия Механизма Perl, но те, кому хватит замороченности, могут попробовать изменить то, как Механизм видит шаблон. Поскольку Perl интерпретирует шаблон аналогично строке в двойных кавычках, можно использовать чудеса перегруженных строковых констант, чтобы избранные нами текстовые последовательности автоматически транслировались в другие.

В примере ниже определяются два преобразования, которые должны происходить, когда Perl обнаруживает шаблон. Во-первых, мы определим метасимвол \tag, чтобы при обнаружении в шаблоне он автоматически транслировался в (?:<.*?>), что соответствует большинству тегов HTML и XML. Во-вторых, мы «переопределим» метазнак \w, чтобы он работал только с буквами английского алфавита.

Мы определим также пакет с именем Tagger, который укроет перегрузку от нашей основной программы. Сделав это, мы сможем сказать:

```
use Tagger;
$_ = "<I>верблюд</I>";
say "Найден верблюд в тегах" if /\tag\w+\tag/;
```

Ниже приводится содержимое файла *Tagger.pm*, оформленное в виде модуля Perl (см. главу 11):

```
package Tagger;
use overload;

sub import { overload::constant 'qr' => \&convert }

sub convert {
    my $re = shift;
    $re =~ s/ \\tag /<.*?>/xg;
    $re =~ s/ \\w   /[A-Za-z]/xg;
    return $re;
}

1;
```

Модуль Tagger получает шаблон непосредственно перед интерполяцией, поэтому обойти перегрузку можно, обойдя интерполяцию, например:

```
$re = '\tag\w+\tag';   # Эта строка начинается с \t, символа табуляции
print if /$re/;        # Ищет символ табуляции, за которым следует "a"...
```

Чтобы преобразовать интерполированную переменную, вызовите функцию convert непосредственно:

```
$re = '\tag\w+\tag';         # Эта строка начинается с \t, символа табуляции
$re = Tagger::convert $re;  # разыменовать \tag и \w
print if /$re/;             # $re становится <.*?>[A-Za-z]+<.*?>
```

Если вы все еще не поняли, что это за штуки sub в модуле Tagger, то скоро поймете, потому что этому посвящена глава 7.

Альтернативные механизмы

Начиная с версии v5.10, в Perl появилась возможность подменять механизм регулярных выражений альтернативными библиотеками сопоставления с шаблонами. Принцип действия этой возможности описывается в странице *perlreapi* справочного руководства. Описание достаточно сложное и предназначено для технически подкованных хакеров.

Однако удача может улыбнуться вам. Возможно, в архиве CPAN уже существуют расширения для Perl, позволяющие подменять механизм регулярных выражений. При их использовании вы пишете свои шаблоны как обычно, и когда приходит время задействовать их, передаете управление альтернативному механизму. В табл. 5.18 перечислены некоторые модули из архива CPAN, позволяющие использовать механизмы регулярных выражений из других языков в своем коде на Perl (по состоянию CPAN на осень 2011 года). К тому времени, когда вы будете читать эти строки, подобных модулей может оказаться намного больше, поэтому ищите внимательнее.

Таблица 5.18. Альтернативные механизмы регулярных выражений

Модуль	Описание	Версия	Обновлен	В текущее время сопровождает
re::engine::LPEG	Механизм регулярных выражений LPEG	0.05	2010-07-09	Франсуа Перра (François Perrad)
re::engine::RE2	Механизм регулярных выражений RE2 Расса Кокса (Russ Cox)	0.08	2011-04-22	Дэвид Ледбитер (David Leadbeater)
re::engine::Plugin	Обобщенный API для разработки собственных механизмов регулярных выражений	0.09	2011-04-05	Винсент Пит (Vincent Pit)
re::engine::Plan9	Механизм регулярных выражений из Plan9!	0.16	2010-03-31	Эвар Арнфьорд Бьярмасон (Ævar Arnfjörð Bjarmason)
re::engine::Oniguruma	Механизм регулярных выражений Oniguruma из Ruby	0.05	2011-07-10	藤吾郎
re::engine::Lua	Механизм регулярных выражений из языка Lua	0.06	2008-12-20	Франсуа Перра (François Perrad)
re::engine::PCRE	Механизм регулярных выражений «Perl-Compatible RegEx» Фила Хазеля (Phil Hazel)	0.17	2011-01-29	Эвар Арнфьорд Бьярмасон (Ævar Arnfjörð Bjarmason)

(Заметили что-нибудь необычное в именах авторов? Имена более двух третей из них даже невозможно записать символами ASCII. Добро пожаловать в XXI век, век глобализации!)

Один из механизмов, а именно – RE2 Расса Кокса, заслуживает особого внимания. Это библиотека на C и C++, которая используется в языке программирования Go и в ряде других мест. Самое интересное, что она поддерживает высочайшую совместимость с Perl, включая отличную поддержку Юникода, и при этом избегает потенциальных ловушек катастрофического снижения производительности в результате действия механизма возвратов. Как это возможно? В отличие от рекурсивного механизма возвратов, используемого в Perl, в RE2 применяется гибридный подход на основе NFA/DFA, который не подвержен серьезной деградации даже в патологических случаях.

Этот механизм можно успешно применять в критических ко времени выполнения приложениях, где необходимо дать пользователю возможность создавать шаблоны, но нельзя допустить, чтобы поиске навечно «завис». Первоначально разработанный для проекта компании Google, Code Search (*http://codesearch.google.com*), где время является существенным фактором, механизм RE2 также используется через свой Perl-интерфейс на сайте *http://grep.cpan.me*. Этот сайт позволяет осуществлять поиск на основе шаблонов по всему архиву CPAN.

Чтобы после установки механизма re::engine::RE2[1] включить его в работу, достаточно просто добавить инструкцию use re::engine::RE2 в лексической области видимости регулярных выражений, для выполнения которых предполагается использовать механизм RE2 вместо механизма Perl. Вот и все.

Ниже приводится пример, где механизм RE2 оставляет далеко позади любой рекурсивный механизм с возвратами. Сначала проведем хронометраж обычного механизма регулярных выражений в Perl:

```
% time perl -E 'say (("a" x 17) =~ /a*a*a*a*a*a*a*a*a*a*[Bb]/i || 0)'
>/dev/null
  1.564u 0.005s 0:01.57
% time perl -E 'say (("a" x 23) =~ /a*a*a*a*a*a*a*a*a*a*[Bb]/i || 0)'
>/dev/null
  17.757u 0.025s 0:17.84
% time perl -E 'say (("a" x 29) =~ /a*a*a*a*a*a*a*a*a*a*[Bb]/i || 0)'
>/dev/null
  127.965u 0.180s 2:09.20
```

А теперь – механизма RE2:

```
% time perl -Mre::engine::RE2 -E 'say (("a" x 500) =~
/a*a*a*a*a*a*a*a*a*a*[Bb]/i || 0)'
>/dev/null
  0.004u 0.002s 0:00.00
% time perl -Mre::engine::RE2 -E 'say (("a" x 5000) =~
/a*a*a*a*a*a*a*a*a*a*[Bb]/i || 0)'
>/dev/null
  0.004u 0.002s 0:00.00
% time perl -Mre::engine::RE2 -E 'say (("a" x 50000) =~
/a*a*a*a*a*a*a*a*a*a*[Bb]/i || 0)'
>/dev/null
  0.004u 0.002s 0:00.00
```

Как видите, при использовании механизма RE2 время выполнения не растет пропорционально объему входной информации, оно напрямую зависит только от размера регулярного выражения. Если входная строка окажется настолько большой, что в ней поместится весь архив CPAN, это преимущество трудно будет не заметить. Да, это несколько надуманный пример, однако шаблоны удивительно часто вызывают проблемы подобного характера.

Модуль re::engine::RE2 можно настроить на использование механизма RE2 для шаблонов, которые он способен будет обработать, и на возврат к использованию встроенного в Perl механизма для тех шаблонов, которые не могут быть им обработаны, что обеспечивает его 100% совместимость. Или, если вы разрабатываете внешнюю службу, можно настроить его на использование только RE2 без возврата к встроенному Механизму, и тем самым исключить риск атаки «отказ в обслуживании».

Дополнительную информацию об архитектуре RE2 и о конечных автоматах в целом можно найти в серии из трех статей Расса Кокса (Russ Cox): «Regular Expression Matching Can Be Simple and Fast», «Regular Expression Matching: The Virtual Machine Approach» и «Regular Expression Matching in the Wild».

---

[1] См. указания в главе 19, если вы еще не знаете, как это сделать.

6

Юникод

Если вы никогда не слышали о Юникоде, вероятно, последние 20 лет вы прожили на необитаемом острове и пользовались механической печатной машинкой. Свое двадцатилетие Юникод отпраздновал в начале 2010 года. Даже если вы слышали об этом стандарте, вы можете не знать, что это такое на самом деле, и как с ним работать. Стыдиться тут совершенно нечего, ведь все до единого люди, включая и его изобретателей, все еще продолжают изучать Юникод. Мы определенно не сможем охватить все нюансы и тонкости Юникода в этой главе и даже в этой книге, но мы надеемся помочь вам начать пользоваться Юникодом в Perl.

Работа с Юникодом в наше время не является вопросом выбора: это насущная необходимость. Значительная часть текстовой информации в Сети хранится в Юникоде,[1] и многие крупные коллекции документов на 100% хранятся в Юникоде. Поскольку веб-браузеры прилагают все усилия, чтобы корректно отображать текстовую информацию, содержащую любые наборы символов, вы, вероятно, и не заметили, сколь часто на практике применяется Юникод. Языки программирования без качественной поддержки Юникода отстали на десятилетия, как и программы, написанные на этих языках. Возможно, они годились для восьмидесятых или даже для девяностых, но сегодня нам необходимы более мощные инструменты.

Так что же нас ждет здесь?

Компьютеры хранят символы в виде чисел. На заре компьютеров то были небольшие целые числа, имеющие длину 5, 6, 7 или 8 битов. Кодировка EBCDIC использовала 8 битов и основывалась на перфокартах. Кодировка ASCII использует только 7 битов, оставляя один бит в каждом байте для других целей – многих, многих других целей, порой противоречащих друг другу.

Так что в те дни практически каждый обитатель западных земель путал символы с маленькими числами в диапазоне от 0 до 127 или от 0 до 255. И хотя чисел получалось больше, чем клавиш на клавиатуре, их не хватало, и люди в разных странах имели собственные представления о том, какие символы должны представлять эти числа.

---

[1] В кодировке Юникода UTF-8, если выражаться точно.

Такого набора букв вполне хватало, чтобы отправить телеграмму на простом английском языке, но было недостаточно для представления всех символов латинского, греческого и кириллического алфавитов, не говоря уже об алфавитах, используемых в Азии. Жителям азиатских стран пришлось определять собственные, взаимно несовместимые 16-битные наборы символов. Было чрезвычайно трудно, а порой и невозможно включить текст на разных языках в один документ, поскольку разные символы из разных алфавитов могли быть представлены одними и теми же числовыми кодами.

Люди привыкли создавать наборы символов, отражающие их потребности в контексте собственной культуры. Во всякой культуре люди естественным образом ленивы, а потому обычно стремились включать только необходимые им символы и исключать ненужные. Такой исключающий подход работает, пока мы общаемся только с людьми, принадлежащими нашей собственной культуре. Но как только мы начали использовать Интернет для обмена информацией между разными культурами, такой подход стал источником проблем. Не так просто разобраться, как с помощью американской клавиатуры набирать латинские символы с акцентами, не говоря уже о более экзотических символах.

В результате появилась идея создать Юникод: единую систему символов, которую можно использовать взаимозаменяемо для набора практически любых текстов. Юникод охватывает не только современные алфавиты, но и множество древних, а также специальные символы, используемые в полиграфии, математике, лингвистике и многих других отраслях, включая даже смайлики, используемые в сотовых телефонах. См. табл. 6.1.

Таблица 6.1. Примеры символов Юникода

Латинские буквы	Å ǽ ç ð é ﬃ ï İ ñ ø œ š ſß ʒ þ
Греческие буквы	α ϕ β Γ γ Δ δ ϝ θ λ Ξ ξ π Σ σ ς φ χ Ψ Ώ ϕ
Кириллические буквы	а Б б д ж з И и к Л с т У у Ф ф Ш ь Э Ю я
Математические буквы	𝒜 𝒟 𝔅 ℜ 𝔼 𝔽 ɪ ŋ ʔ ξ ς 𝚯 𝚿 𝛀
Математические символы	÷ × ± ≤ ≠ ≈ ≡ ∅ ∈ ∛ ∆ Ω ⊆ ∞ ∫ ∂ ∴ ℵ₀
Символы валют	¢ $ £ ₤ € ₵ ₠ ₣ ƒ ₱ ₩ ¥
Точки	. · · ○ ▪ ● ‥ … ⋯
Дефисы и тире	- - ▬ — - - – —
Кавычки	‘ “ ‘ ’ “ ” ‹ › « » ⌈ ⌋

Как видите, некоторые из них очень похожи между собой. Юникод различает символы не по внешнему виду, а по их назначению. В первую очередь Юникод – это семантический код, и только потом – код начертания, и то лишь потому, что начертание является частью семантики. В отличие от кодировки ASCII, охватывающей небольшой набор символов и определяющей малое количество свойств для этих символов, Юникод охватывает более обширные сферы. Юникод можно считать расширенной формой ASCII с огромным количеством символов. Но Юникод – это не только более обширный набор символов, это также свод правил распределения символов по категориям и работы с ними.

Помимо символов и их свойств, Юникод определяет также: порядок отображения регистров символов и приведения символов к единому регистру (свертки); комби-

нирующие символы; группы графем; формы нормализации; порядок сопоставления; свойства символов для использования при поиске по шаблону, включая имена символов, категории и алфавиты; числовые эквиваленты (например, позволяющие определить, что число, обозначаемое символом U+216B, «XII», имеет значение 12); ширину печати; двунаправленность; правила разрыва слов и строк; и вариации начертания. Список далеко не полон...

Первая предварительная поддержка Юникода появилась в Perl v5.6, однако важные задачи ввода/вывода нам удалось решить только к Perl v5.8. К версии v5.12 мы решили большинство более мелких проблем, и, начиная с версии v5.14, Perl обеспечивает полную и своевременную поддержку стандарта Юникода версии v6.0 – теперь вы можете использовать Юникод в Perl, не прибегая к каким-либо ухищрениям. Почти. Во всяком случае, в Perl этих ухищрений по-любому потребуется меньше, чем в любом другом языке.

Мы хотим сказать, что сохранили возможность решать простые задачи простым способом, при этом не сделав сложные неразрешимыми. Первая простая вещь, на которую следует обратить внимание, – Perl позволяет хранить в строках любые символы, с любыми значениями кодов. Кодировка ASCII ограничивает код символа 7 битами, Latin-1 – 8 битами, а Юникод поддерживает коды длиной до 21 бита.[1] Но символы в Perl не ограничены столь маленькими числами. В настоящее время на 64-битных архитектурах символы в Perl ограничены 64-битными значениями, но даже с учетом этого ограничения Perl позволяет представлять на 18 446 744 073 708 437 504 кодов символов больше, чем имеется в Юникоде. (Мы ведь сказали «с любыми значениями кодов»? Так-то.)

В Юникоде каждому символу соответствует уникальный номер, который называется *кодом символа*. Отсюда и происходит название Юникод: уникальный, универсальный код для каждого самостоятельного символа. Например, символ с именем LATIN CAPITAL LETTER A имеет порядковый десятичный номер 65, шестнадцатеричный 0x41. Он часто записывается как U+0041; существующие соглашения таковы, что префикс «U+» обозначает не просто число, а число, представляющее код символа Юникода.

Если вы когда-нибудь принимали по ошибке «l» за «1», «0» за «O» или «,» за «.», то уже знаете, как человек легко может спутать символы. Вы также знаете, что компьютер никогда не путается. Для него не имеет значения, как выглядит начертание символа в шрифте. Для него важно лишь назначение символа. Например, в табл. 6.2 перечислены символы, которые выглядят почти одинаково при использовании почти любых шрифтов.

В первой колонке табл. 6.2 приводится начертание символа. Вы сможете использовать литералы Юникода, подобные этим, в своем программном коде, только если в текущей лексической области видимости действует прагма `use utf8`. Большинство современных текстовых редакторов и оконных систем предоставляет различные способы ввода таких символов, однако они могут быть отключены по умолчанию, поэтому может потребоваться провести небольшие исследования, если вы не слишком ленивы. Но постойте, вам может пригодиться и такой способ: если вы не знаете, как ввести тот или иной символ, его можно попробовать

---

1 На деле эта длина составляет лишь 20,087463 бита, поскольку Юникод содержит лишь 0x110000 кодов, а не 0x200000.

отыскать где-нибудь, выделить мышью, скопировать в буфер обмена и вставить в свой текст.

Таблица 6.2. Символы Юникода, которые легко перепутать с буквой А

Начертание	Код	Категория	Алфавит	Имя
A	U+00041	Lu	Latin	LATIN CAPITAL LETTER A
Ａ	U+0FF21	Lu	Latin	FULLWIDTH LATIN CAPITAL LETTER A
Α	U+00391	Lu	Greek	GREEK CAPITAL LETTER ALPHA
А	U+00410	Lu	Cyrillic	CYRILLIC CAPITAL LETTER A
𝖠	U+1D5A0	Lu	Common	MATHEMATICAL SANS-SERIF CAPITAL A
𝙰	U+1D670	Lu	Common	MATHEMATICAL MONOSPACE CAPITAL A

Если отвлечься от внешнего вида, начертание, в некотором смысле, является наименее полезным аспектом символа, потому что невозможно знать, как тот или иной символ отображается (и отображается ли вообще) с применением какого-либо другого шрифта, кроме того, что используете вы сами. У вас начертание может выглядеть очень неплохо, но не верьте своим глазам – верьте числам. Во второй колонке указаны числовые коды символов, в стандартном формате Юникода. А вот несколько способов работать с кодами символов Юникода в языке Perl:

```
if (ord($somechar) == 0x0391) { .... }
$alpha = "\x{391}";
$alpha = "\N{U+391}";
$alpha = chr(0x391);
```

Двумя наиболее важными свойствами символов, помимо их числовых кодов, являются: принадлежность к категории (колонка «Категория» в табл. 6.2) и принадлежность к алфавиту (колонка «Алфавит» в табл. 6.2). В шаблонах свойства кодов символов Юникода чаще всего используются в роли именованных классов символов, где \p{*PROPERTY*} соответствует любому коду символа с данным свойством, а \P{*PROPERTY*} соответствует любому коду символа, не обладающему данным свойством.

```
/^\p{GC=Lu}+$/                       # Все заглавные буквы
/^\p{script=Greek}+$/                # символы греческого алфавита
/[\P{script=Latin}\P{script=Common}]/ # не пересекаются
```

Последний шаблон соответствует строкам, содержащим любые коды символов, не принадлежащие алфавитам Latin и Common. Поскольку регистр символов, наличие пробелов и символов подчеркивания в именах свойств не имеет большого значения, вы можете форматировать их как вам заблагорассудится. То есть, если вам покажется, что \p{gc=modifier_letter} читается лучше, когда используются только символы нижнего регистра, а \P{SC=INHERITED} – когда используются только символы верхнего регистра, пишите как нравится: Perl об этом не беспокоится. Или поступите наоборот, если вам того захочется.

Помимо показанных выше свойств, состоящих из двух частей, для обозначения категорий и алфавитов Perl предлагает также односоставные псевдонимы, что позволяет написать просто \p{Lu} и \p{Greek}. Например, если вам потребуется убедиться, что строка содержит только символы из алфавитов Latin и Greek, достаточно выполнить следующую проверку:

```
$mylang = qr/[\p{Latin}\p{Greek}\p{Common}\p{Inherited}]/;
if ($string =~ /\A$mylang+\z/) { ... }
```

Мы добавили псевдоним Common для обозначения кодов символов, общих для различных алфавитов, таких как цифры и знаки пунктуации, а псевдоним Inherited – для обозначения комбинационных кодов символов (как правило, диакритических знаков), которые присоединяются к алфавиту символа, с которым используются. Комбинационные коды символов не имеют аналогов в ASCII, и, случается, сбивают с толку тех говорящих на ASCII, кто впервые знакомится с Юникодом. Пожалуй, наиболее близким понятием в ASCII является перечеркивание с использованием символа забоя, за исключением того, что в Юникоде комбинационный код автоматически применяется к предшествующему базовому коду. Подробнее о них рассказывается ниже, в разделе «Графемы и нормализация».

Вы могли заметить, что в этой главе мы проводим различие между «символом» и «кодом» или «кодом символа». В других местах (включая другие главы этой книги) данные термины часто используются взаимозаменяемо, и термин «символ» также часто используется вместо термина «графема», но в данной главе нам требуется чуть большая точность. Коды – это целые числа, составляющие строку, когда она считается списком целых чисел, тогда как «символ» – достаточно расплывчатый термин, который в человеческом понимании может означать и код (или кодовый пункт), и графему. В общем случае следует помнить, что слово «символ» в разговоре может иметь до трех-четырех различных смыслов.

Последняя колонка в табл. 6.2 содержит имена символов. Ой, т.е. названия кодов символов. Чтобы получить возможность называть коды по именам в тексте программы, имена сначала нужно загрузить при помощи модуля charnames, а потом записывать в форме \N{...}, как показано ниже:

```
use charnames qw(:full);

$alpha      =     "\N{GREEK CAPITAL LETTER ALPHA}";
$alpha_code = ord "\N{GREEK CAPITAL LETTER ALPHA}";
if ($string =~    /\N{GREEK CAPITAL LETTER ALPHA}/) { ... }
```

Пользоваться названиями кодов символов намного правильнее, чем их числовыми значениями, поскольку имена делают текст программы более понятным.

Другие интересные приемы, основанные на использовании формы записи \N{...}, приводятся в разделе «charnames» главы 29.

Не рассказывай, а показывай

Если картина стоит тысячи слов, то возможность вставлять в программу те или иные символы определенно стоит не меньше пятидесяти. Поэтому для начала необходимо сообщить Perl, что текст вашей программы содержит символы Юникода, а не простые байты.[1] Вас никто не обязывает делать это, но данный шаг немного облегчит жизнь, если вам потребуется вставить в программный код символы Юникода.

[1] Возможно, вы предпочитаете называть их «октетами»; пусть так, но мы считаем эти два слова близкими синонимами, поэтому мы будем придерживаться терминологии, более близкой технарям.

Perl по сей день предполагает, что все модули с исходными текстами содержат только символы ASCII, если явно не указано иное (хотя мы осознаем, что рано или поздно кодировкой по умолчанию станет Юникод). Вы всегда можете определять кодовые пункты Юникода с помощью приемов, упомянутых выше, но литералы всегда будут интерпретироваться как отдельные байты. Встретив литерал символа в кодировке UTF-8, Perl не посчитает его одним логическим символом, а будет рассматривать его как один, два, три или даже четыре отдельных символа, с порядковыми номерами, не превышающими 256. Чтобы этого не случилось, используйте следующие объявления:

```
use v5.14;      # включает механизм unicode_strings
use utf8;       # обрабатывает литералы в кодировке UTF-8
```

Первое гарантирует, что кодовые пункты с порядковыми номерами в непростом диапазоне 128–255 будут интерпретироваться как строки Юникода, а второе сообщает компилятору Perl, что файл в целом содержит текст Юникода в кодировке UTF-8. Прагма `utf8` **позволяет использовать Юникод в литералах строк и регулярных выражений.**

```
my $dwarf   = "Þórinn Eikinskjaldi";
my $search  = "búsqueda";
my $measure = "Ångström";
my $how     = "à contre-cœur";
my $motto   = "👪💗🐪";
```

Такой текст намного проще читается, хотя набрать его может оказаться не так просто, как вот этот эквивалент:

```
use charnames qw(:full);

my $dwarf   = "\N{LATIN CAPITAL LETTER THORN}\N{LATIN SMALL LETTER
               O WITH ACUTE}rinn Eikinskjaldi";
my $search  = "b\N{LATIN SMALL LETTER U WITH ACUTE}squeda";
my $measure = "A\N{COMBINING RING ABOVE}ngstro\N{COMBINING DIAERESIS}m";
my $how     = "\N{LATIN SMALL LETTER A WITH GRAVE} contre-c\N{LATIN
               SMALL LIGATURE OE}ur";
my $motto   = "\N{FAMILY}\N{GROWING HEART}\N{DROMEDARY CAMEL}";
```

При этом оба способа, представленные выше, предпочтительнее использования секретных кодов в тексте программы:

```
my $dwarf   = "\x{DE}\x{F3}rinn Eikinskjaldi";
my $search  = "b\x{FA}squeda";
my $measure = "\x{C5}ngstr\x{F6}m";
my $how     = "\x{E0} contre-c\x{153}ur";
my $motto   = "\x{1F46A}\x{1F497}\x{1F42A}";
```

Но и это еще не все. В области действия прагмы `utf8` **появляется возможность использовать Юникод в идентификаторах Perl.**

```
# несколько наборов символов
my @IsO   = qw( Latin1 Latin2 Latin15 );
my @µsoft = qw( cp852 cp1251 cp1252 );
my @鯉    = qw( koi8-f koi8-u koi8-r );
```

```
# включать ли ответы, которые не возвращают результатов
my $INCLUÍR_NINGUNOS = 0;

# имеют ли значение диакритические знаки
my $SI_IMPORTAN_MARCAS_DIACRÍTICAS = 0;

# считайте << за оператор "имеет" .)
my @ciudades_españolas = ordenar_a_la_española(<<'LA_ÚLTIMA' =~ /\S.*\S/g);
....
....
LA_ÚLTIMA

my $déjà_imprimée:      # название города

# Идентификатор на греческом языке
my @ὑπέρμεγας = ( );

# А теперь мы просто выпендриваемся :-)
my $ʇndʇno = uʍopəpısdn($input);
```

На практике, если вы будете использовать идентификаторы не на английском языке, вам, вероятно, захочется писать комментарии на соответствующем языке. Perl упрощает возможность писать программы на собственном языке для людей во всем мире, не вынуждая их изучать английский язык.

В настоящее время вы должны ограничить применение Юникода только приватными переменными. Это обусловлено особенностями хранения глобальных переменных, а также особенностями отображения имен модулей в имена файлов в локальной файловой системе. Первое ограничение, как ожидается, будет снято в ближайшем будущем, а вот второе пока остается предметом для исследования.

Доступ к данным в Юникоде

Для целей внутреннего хранения любых кодов символов Perl применяет формат, совместимый с Юникодом. То есть 21 младший бит в точности соответствует диапазону кодов Юникода, так же как 8 младших битов Юникода соответствуют кодировке Latin-1. Как фактически хранятся коды символов, не так важно для рядового пользователя Perl.

При этом, как только возникает необходимость взаимодействовать с внешним миром, приходится предусматривать соответствующую интерпретацию получаемых данных и генерировать выходные данные в формате, приемлемом для принимающей их программы. Внутри Perl символы *декодируются* из внешнего представления в абстрактные символы, но когда требуется вывести эти символы, их необходимо *закодировать* в некоторое ожидаемое представление. Если забыть сделать это, программа сгенерирует то, что иногда называют «широкими символами» (wide characters) или «неправильно сформированными символами UTF-8» (malformed UTF-8 character).

В Perl имеется два основных способа определить кодировку для всего потока данных, плюс различные сокращения, упрощающие жизнь. Если поток ввода/вывода уже открыт, его кодировку можно установить, передав ее в качестве второго аргумента функции `binmode`:

```
binmode(STDIN, ":encoding(CP1252)")
    || die "can't binmode to cp1252: $!"
binmode(STDOUT, ":encoding(UTF-8)")
    || die "can't binmode to UTF-8: $!"
```

Если поток еще не был открыт, кодировку можно назначить вместе с режимом при вызове функции open.

```
open(OUTPUT, "> :raw :encoding(UTF-16LE) :crlf", $filename)
            or die "can't open $filename: $!";
print OUTPUT for @stuff;
close(OUTPUT) or die "couldn't close $filename: $!";
```

При выводе данных фильтр :crlf выполняет преобразование символов \n в пары \r\n; при вводе – наоборот. Этот режим действует по умолчанию в Windows, при работе с текстовыми файлами, но, если это необходимо, его следует явным образом включать в UNIX. См. страницу *PerlIO* справочного руководства, где приводится дополнительная информация о фильтрах ввода/вывода.

Помимо \n и \r\n в Юникоде существуют и другие символы завершения строк. В настоящее время таких последовательностей в Юникоде существует восемь– семь перечисленных ниже символов, плюс последовательность \r\n, состоящая из двух кодов:

```
U+000A LINE FEED (LF)
U+000B LINE TABULATION
U+000C FORM FEED (FF)
U+000D CARRIAGE RETURN (CR)
U+0085 NEXT LINE (NEL)
U+2028 LINE SEPARATOR
U+2029 PARAGRAPH SEPARATOR
```

В Perl нет специального фильтра обобщенной обработки последовательностей завершения строки Юникода, но, если вы можете позволить себе прочитать файл целиком в память, то легко сможете преобразовать все такие последовательности в символы перевода строки:

```
$complete_file =~ s/\R/\n/g;
```

Или разбить содержимое файла на список строк, не имеющих последовательностей завершения строки:

```
@lines = split(/\R/, $complete_file);
```

Чтобы назначить кодировку для вновь открываемых файлов, можно воспользоваться прагмой open. Например, ниже показано, как организовать использование кодировки UTF-8 для всех файлов, которые будут играть роль стандартных потоков ввода/вывода – STDIN, STDOUT и STDERR – если при открытии кодировка не была указана явно:

```
use open qw( :encoding(UTF-8) :std )
```

Если вам достаточно назначить кодировку UTF-8 для стандартных потоков, можно при необходимости пользоваться ключом командной строки -CS или устанавливать переменную среды PERL_UNICODE в значение "S". Если вместо "S" использовать значение "D", все дескрипторы будут открываться в текстовом режиме и с кодировкой UTF-8 по умолчанию. См. главу 17.

Использование ключа командной строки -C или переменной среды PERL_UNICODE требует явного вызова функции binmode для двоичных потоков даже в программах для UNIX, что обычно требуется только в Windows или в переносимых программах. Это может нарушить работу уже имеющихся программ для UNIX, предполагающих, по умолчанию, что они работают с двоичными данными, а не с текстом. Но это не нарушит работу существующих программ, которые не умеют декодировать текст в кодировке UTF-8.

Ниже перечислены механизмы назначения кодировки для потока ввода/вывода, следующие в порядке убывания приоритета (механизм, расположенный выше, может переопределить настройки, выполненные с помощью механизма, расположенного ниже в списке):

1. Явный вызов binmode для уже открытого дескриптора.
2. Включение фильтра во второй аргумент функции open.
3. Прагма open.
4. Ключ командной строки -C.
5. Переменная среды PERL_UNICODE.

Единственным исключением является дескриптор DATA. На него не действуют прагмы use utf8 и open, поэтому при необходимости вам придется указывать кодировку вручную:

```
binmode(DATA, ":encoding(UTF-8)");
```

Из-за особенностей реализации уровней кодирования utf8 и UTF-8, обычно они не возбуждают исключения, встречая неправильно сформированные входные данные. Чтобы исправить ситуацию, добавьте в свой код следующую строку:

```
use warnings FATAL => "utf8";
```

В Perl (начиная с версии v5.14) существует три подгруппы предупреждений, составляющие группу предупреждений "utf8". Эти типы предупреждений бывает полезно различать:

nonchar

66 кодов символов Юникода считаются «несимвольными». Все они относятся к общей категории Unassigned (Cn) и никогда не будут сопоставлены какимлибо символам. Они не могут использоваться в открытом обмене, поэтому программный код может добавлять их в символьные данные в качестве различных сигнальных меток, и такие метки всегда можно отличить от основных данных. 32 несимвольных кода занимают диапазон от U+FDD0 до U+FDEF, а еще 34 располагаются парами в конце каждого сегмента (их шестнадцатеричные коды заканчиваются, соответственно, последовательностями FFFE или FFFF). В некоторых ситуациях может потребоваться разрешить присутствие таких несимвольных кодов, например, для организации взаимодействия между процессами, совместно использующими одни и те же сигнальные маркеры. В таких случаях используйте:

```
no warnings "nonchar";
```

surrogate

Эти кодовые пункты зарезервированы для использования в кодировке UTF-16. На практике нет никаких причин разрешать их использование, и никакие

совместимые процессы не смогут обмениваться ими, потому что программы, использующие кодировку UTF-16, не способны обрабатывать их (хотя, программы, использующие кодировки UTF-8 и UTF-32, могли бы использовать их при желании).

non_unicode

Максимально допустимый код символа в Юникоде имеет значение U+10FFFF, но Perl способен символы с любыми кодами, вплоть до максимального беззнакового целого, поддерживаемого аппаратной архитектурой. В зависимости от различных настроек и фазы Луны, Perl может предупредить о попытке ввода или вывода кодов, выходящих за верхнюю границу (используя категорию предупреждений «non_unicode», которая является подкатегорией «utf8»). Например, "uc(0x11_0000)" вызовет такое предупреждение, вернув входной параметр в качестве результата, так как кодовым пунктом верхнего регистра любого кода символа, не принадлежащего Юникоду, является этот же код.

Из перечисленных подгрупп самой полезной является содержащая коды, выходящие за границы Юникода, и вы наверняка пожелаете разрешить использование таких символов для внутренних нужд.

```
no warnings "non_unicode";
```

Вот одно из возможных применений. То, что следующая операция делает для ASCII:

```
tr[\x00-\x7F][\x80-\xFF];
```

операция ниже делает для Юникода:

```
tr[\x{00_0000}-\x{10_FFFF}][\x{20_0000}-\x{30_FFFF}];
```

Она отображает все коды из допустимого диапазона в соответствующие им символы из недопустимого диапазона. Зачем это может понадобиться? Это один из способов пометить текст, который не должен затрагиваться поиском. Только не забудьте выполнить обратное преобразование, завершив поиск.

Модуль Encode

Стандартный модуль Encode используется чаще неявно, чем явно. Он загружается автоматически всякий раз, когда функции binmode или open передается аргумент :encoding(*ENC*).

Но иногда приходится иметь дело с фрагментами кодированных данных, источником которых не служит поток ввода с назначенной кодировкой, поэтому возникает необходимость декодировать их вручную, прежде чем приступать к их обработке. Такие кодированные строки могут поступать в программу из разных источников за ее пределами, – скажем, из переменных среды, аргументов командной строки, параметров CGI или полей базы данных. Увы, иногда вам даже придется сталкиваться с «текстовыми» файлами, содержащими строки сразу в нескольких кодировках. Вы обязательно столкнетесь с *кракозябрами*.

Во всех этих ситуациях вам придется обратиться к модулю Encode, чтобы реализовать явное кодирование и декодирование данных. Чаще всего вам придется использовать функции (сюрприз!) encode и decode из этого модуля. Если у вас имеются необработанные внешние данные, хранящиеся в некоторой кодированной

форме в виде байтов, передайте их функции decode и превратить в абстрактные символы. С другой стороны, если у имеются некоторые абстрактные символы, и необходимо преобразовать их в некоторую кодированную форму, воспользуйтесь функцией encode.

```
use Encode qw(encode decode);
$chars = decode("shiftjis", $bytes);
$bytes = encode("MIME-Header-ISO_2022_JP", $chars);
```

Например, если известно, что терминал настроен на работу с кодировкой UTF-8, декодировать содержимое @ARGV можно следующим образом:

```
# действует так же, как perl -CA
if (grep /\P{ASCII}/ => @ARGV) {
    @ARGV = map { decode("UTF-8", $_) } @ARGV;
}
```

Если ваша рабочая среда не предлагает повсеместную поддержку кодировки UTF-8, не следует исходить из предположения, что терминал всегда настроен на использование кодировки UTF-8. Для него может быть установлена национальная кодировка. Стандартный модуль Encode не поддерживает операции, чувствительные к региональным настройкам, однако в архиве CPAN имеется модуль Encode::Locale, поддерживающий такие операции. Используйте его следующим образом:

```
use Encode;
use Encode::Locale;

# использовать "locale" как аргумент encode/decode
@ARGV = map { decode(locale => $_) } @ARGV;

# или как поток для binmode или open
binmode $some_fh, ":encoding(locale)";

binmode STDIN,  ":encoding(console_in)"  if -t STDIN;
binmode STDOUT, ":encoding(console_out)" if -t STDOUT;
binmode STDERR, ":encoding(console_out)" if -t STDERR;
```

Базы данных – еще одна область, где может пригодиться возможность выполнять кодирование и декодирование вручную. Конкретика, впрочем, зависит от используемой системы управления базами данных. Если речь о простых DBM-файлах, нижестоящая библиотека работает с байтами, а не строками кодов символов, поэтому текст Юникода нельзя записать непосредственно в файл DBM. Если попытаться сделать это, программа возбудит исключение Wide character in subroutine. Чтобы сохранить пары ключ/данные в DBM-хеше %dbhash, их нужно сначала преобразовать в кодировку UTF-8:

```
use Encode qw(encode decode);

# предполагается, что $uni_key и $uni_value - строки
# абстрактных символов Юникода

$enc_key   = encode("UTF-8", $uni_key);
$enc_value = encode("UTF-8", $uni_value);
$dbhash{$enc_key} = $enc_value;
```

Отсюда следует, что обратная операция извлечения значения в Юникоде требует сначала закодировать ключ перед его использованием, а затем декодировать извлеченное значение:

```
use DB_File;
use Encode qw(encode decode);

tie %dbhash, "DB_File", "pathname";

# $uni_key хранит обычную строку Perl (абстрактные символы Юникода)
$enc_key   = encode("UTF-8", $uni_key);

$enc_value = $dbhash{$enc_key};
$uni_value = decode("UTF-8", $enc_value);
```

После этого полученную строку $uni_value можно использовать как любую другую строку Perl. Хотя перед этим она была лишь последовательностью байтов – целых чисел, хранящихся в форме строки. (И эти целые числа ничем *не напоминали* коды символов Юникода.)

Начиная с версии v5.8.4 появилась возможность использовать стандартный модуль DBM_Filter для прозрачного кодирования/декодирования данных.

```
use DB_File;
use DBM_Filter;

use Encode qw(encode decode);

$dbobj = tie %dbhash, "DB_File", "pathname";
$dbobj->Filter_Value("utf8");

# $uni_key содержит обычную строку Perl (абстрактные символы Юникода)
$uni_value = $dbhash{$uni_key};
```

Ошибочные представления о регистре

Если вы знакомы только с набором символов ASCII, практически все ваши предположения относительно регистра символов Юникода будут ошибочны. В ASCII существуют буквы верхнего регистра и буквы нижнего регистра, но в Юникоде существует еще третий регистр – заглавный. Этот регистр не имеет широкого использования в английском языке, но применяется в других системах письменности, основанных на латинском или греческом алфавите.

Обычно заглавный регистр совпадает с верхним регистром, но не всегда. Он используется, когда только первая буква должна быть большой. Некоторые символы Юникода выглядят как две буквы, напечатанные рядом друг с другом, но в действительности представлены одним кодом символа. В слове, где только первая часть может быть большой, но не остальные, заглавный регистр позволит выделить размером лишь установленную часть. Эта возможность существует в основном для поддержки устаревших кодировок, и в настоящее время чаще применяются коды символов, для которых заглавный регистр отображается в два различных кода, один в верхнем регистре и один в нижнем. Ниже демонстрируется один из таких устаревших символов:

```
use v5.14;
use charnames qw(:full);
my $beast = "\N{LATIN SMALL LETTER DZ}ur";
say for $beast, ucfirst($beast), uc($beast);
```

Он выведет три слова: «dzur», «Dzur» и «DZUR», каждое из которых состоит из трех кодов.

Некоторые буквы не имеют регистра, а некоторые небуквы имеют. Регистр букв – относительно редкое явление в системах письменности. Лишь восемь алфавитов из почти 100, поддерживаемых Юникодом версии v6.0, имеют регистровые символы: Armenian, Coptic, Cyrillic, Deseret, Georgian, Glagolitic, Greek и Latin, плюс некоторые из алфавитов Common и Inherited.

При изменении регистра символов может измениться и длина строки. При *простом изменении регистра* (*simple casemapping*) измененная строка всегда имеет ту же длину, что исходная, но при *полном изменении регистра* это совершенно не обязательно. Например, строка «tschüß» после преобразования в верхний регистр «TSCHÜSS» становится на один символ длиннее.

Разные строки в одном регистре могут отображаться в одну и ту же строку в другом регистре. Обе сигмы греческого алфавита, «σ» и «ς», отображаются в одну и ту же букву верхнего регистра, «Σ», и это лишь самый простой пример. Чтобы достаточно разумно (или, хотя бы, не так безумно) решить проблему с подобными превращениями, было введено понятие свертки регистра, применяемой для сравнения символов без учета их регистра. Если после свертки регистров символов в двух строках получаются одинаковые строки, они *по определению* считаются эквивалентными без учета регистра символов.

Сопоставление без учета регистра символов в Perl всегда поддерживалось модификатором шаблонов `/i`, который сравнивает результаты *свертки регистров*. Начиная с версии v5.16, поддерживается функция `fc`, позволяющая сравнивать результаты свертки регистров двух строк, чтобы определить, являются ли они эквивалентными без учета регистра символов. До версии v5.16 функция `fc` была доступна в модуле `Unicode::CaseFold` из архива CPAN.

Загляните в свою копию страницы *perlfunc* справочного руководства и примечания к выпуску версии (*perldelta*), чтобы определить, поддерживает ли ваша версия Perl эту возможность. Если такая поддержка имеется, значит, наряду с ней поддерживается интерполируемая экранированная последовательность `\F`, действующая подобно `\L` и `\U`, но производящая свертку регистра.

Еще одна интересная особенность Юникода, несвойственная ASCII, заключается в том, что операция преобразования регистра может оказаться необратимой. Например, `lc("ẞ")` возвращает `"ß"`, но `uc("ß")` возвращает `"SS"`, а `lc("SS")` – `"ss"`, что далеко от первоначальной формы. Но не обязательно прибегать к экзотическим двухсимвольным комбинациям, чтобы показать, что обратимость операции преобразования регистра не гарантируется. Вспомните греческие сигмы: «σ» – это обычная форма, а «ς» – форма, используемая в конце слов, и для обеих форм имеется единственная форма в верхнем регистре, «Σ». Если выполнить преобразование `lc(uc("ς"))`, результат будет отличаться от первоначального значения «ς» – вы получите символ «σ».

Однако не все символы, имеющие регистр, способны изменять его. В Юникоде символ *не обязан иметь* верхний или заглавный регистр лишь потому, что он

имеет нижний регистр. Например, uc("M^{c}Kinley") вернет "M^{c}KINLEY", потому что символ MODIFIER LETTER SMALL C имеет нижний регистр, но не имеет других регистров, иначе он выглядел бы неправильно. Аналогично капитель, фактически, представлена символами нижнего регистра, потому что по высоте они соответствуют строчным буквам. В строке «Boulder Camera» первая буква каждого слова находится в верхнем регистре, а остальные – в нижнем.

Не все символы, имеющие нижний регистр, являются буквами. Регистр – это свойство, не зависящее от принадлежности к той или иной категории. Римские цифры, например, имеют регистр, сравните «VIII» и «viii». Существуют даже буквы, считающиеся буквами нижнего регистра, для которых выполняется равенство GC=Lm и не выполняется GC=Ll.

В случае ASCII, чтобы перевести слово в регистр «заголовка», используется функция ucfirst(lc($s)), которая, однако, не гарантирует корректную работу с Юникодом, потому что перевод в заглавный регистр из нижнего регистра не всегда дает тот же результат, что и перевод в заглавный регистр оригинала. Это замечание верно и для других комбинаций. Правильный способ состоит в том, чтобы первую букву отдельно перевести в заглавный регистр, а остальные – в нижний, либо явно вызвав соответствующие функции, либо с помощью операции подстановки:

```
$tc = ucfirst(substr($s, 0, 1)) . lc(substr($s, 1));

s/(\w)(\w*)/\u$1\L$2/g;
```

Помимо основных категорий (General Categories) в Юникоде имеется довольно много категорий, связанных с регистром символов. В табл. 6.3 перечислены категории, поддерживаемые Юникодом версии v6.0. Все они являются логическими свойствами, поэтому допускается использовать форму записи с одним элементом. Иначе говоря, для проверки наличия свойства в любом кодовом пункте вместо \p{CWCM=Yes} и \p{CWCM=No} можно использовать форму записи \p{CWCM} и \P{CWCM}, соответственно.

Таблица 6.3. Свойства, связанные с регистром

Короткое	Длинное
Cased	Cased
Lower	Lowercase
Title	Titlecase
Upper	Uppercase
CWL	Changes_When_Lowercased
CWT	Changes_When_Titlecased
CWU	Changes_When_Uppercased
CWCM	Changes_When_Casemapped
CWCF	Changes_When_Casefolded
CWKCF	Changes_When_NFKC_Casefolded
CI	Case_Ignorable
SD	Soft_Dotted

Свойства Lower и Upper соответствуют всем кодам символов, обладающих соответствующими свойствами, не только буквам. В настоящее время не существует небуквенных символов, имеющие заглавный регистр, поэтому Title (пока) суть то же самое, что и gc=Lt. Однако в области действия модификатора /i все три свойства соответствуют свойству Cased, которым обладают не только буквы. При поиске без учета регистра символов эквивалентом gc=Lt является свойство Case_Letter.

Графемы и нормализация

Выше уже упоминались символы, такие как LATIN SMALL LETTER DZ, которые представлены одним кодом, но выглядят как два символа. Однако гораздо чаще встречается противоположная ситуация. То есть, для отображения одиночного символа (*графемы*) может потребоваться более одного кода. Представьте букву с диакритическим знаком (или парой знаков), скажем, «é» в слове «résumé». Каждая такая буква может обозначаться одним кодом или двумя. Может даже так случиться, что одна буква «é» в слове будет обозначена единственным кодом буквы, а другая – кодом буквы, за которым следует код диакритического знака. Глядя только на изображения символов, невозможно заметить разницу, потому что они считаются эквивалентными. Эта тонкость имеет серьезные последствия для обработке практически любого текста, и она почти противоречит тому, что говорилось выше о неважности начертаний. В данном конкретном смысле начертания приобретают особую важность.

Комбинационные символы используются, например, для превращения «n» в «ñ», «c» – в «ç», «o» – в «ô» или «u» – в «ǘ». В первых трех случаях требуется по одному комбинационному знаку, а в последнем – два. На практике количество комбинационных знаков не ограничивается. Вы можете добавить любое их количество, и в результате создать символы, прежде никем не виданные.

Все это требует серьезного переосмысления и переделки самого разнообразного программного обеспечения. Только представьте, какие функции возлагаются на систему управления шрифтами. (Нет, отказ не принимается.) Возможно, ваши собственные программы обработки текста нуждаются в серьезной переделке. Даже такие простые понятия, как перестановка символов строки в обратном порядке, становится большой проблемой, ведь если переупорядочивать коды, а не графемы, комбинационные знаки окажутся присоединены к другим базовым символам. Niño, María и François превратятся в этом случае в õniN, áiraM и siọcnarF.

Рассмотрим графему, сконструированную на базе алфавитного кода, A, за которым следуют два комбинационных знака, назовем их X и Y. Имеет ли значение порядок, в котором они применяются? Являются ли графемы AXY и AYX одним и тем же символом? Иногда да, иногда нет. В таких графемах, как «ǭ», порядок не имеет значения, поскольку один знак располагается над символом, а второй – под ним. Так как порядок не имеет значения, программа должна считать эту графему за символ LATIN LETTER SMALL O, за которым следуют, в любом порядке, знаки COMBINING OGONEK и COMBINING MACRON. Но иногда оба комбинационных знака располагаются в одной и той же области символа, и тогда порядок *имеет значение*. Подробнее об этом чуть ниже.

Но и этим наши мучения не заканчиваются. Юникод содержит предварительно скомпонованные символы, назначение которых – обеспечивать двунаправленную

совместимость с устаревшими наборами символов. Например, алфавит Latin насчитывает порядка 500 таких символов,алфавит Greek – около 250. Подобные символы во множестве существуют и в других алфавитах.

Например, символу «é» соответствует код U+00E9, LATIN LETTER SMALL E WITH ACUTE. Да, всего один. А сложность вот в чем: обращаться с ним следует так же, как если бы графеме соответствовал код LATIN LETTER SMALL E, за которым следует COMBINING ACUTE ACCENT.

Для графем, включающих более одного комбинационного знака, количество вариантов представления еще больше, так как некоторые из них могут начинаться с того или иного предварительно скомпонованного символа, уже имеющего встроенный комбинационный знак, за которым следует еще один комбинационный знак.

Чтобы управиться со всем этим многообразием, Юникод вводит конкретную процедуру *нормализации*. Согласно глоссарию Юникода по адресу http://unicode.org/glossary/, нормализация – это «устранение из текстовых данных альтернативных представлений эквивалентных последовательностей и преобразование их в формы, эквивалентность которых может быть установлена на двоичном уровне». Иными словами, нормализация позволяет обеспечить уникальную идентификацию для каждой семантической единицы, благодаря чему устраняется связь «один-ко-многим».

Ниже перечислены четыре формы нормализации в Юникоде:

- форма нормализации D (Normalization Form D, NFD), получаемая в результате канонической декомпозиции;
- форма нормализации C (Normalization Form C, NFC), получаемая в результате канонической декомпозиции, за которой следует каноническая композиция;
- форма нормализации KD (Normalization Form KD. NFKD). получаемая в результате совместимой декомпозиции;
- форма нормализации KC (Normalization Form KC, NFKC), получаемая в результате совместимой декомпозиции, за которой следует каноническая композиция.

Обычно предпочтение отдается каноническим формам, потому что при нормализации в совместимые формы может теряться часть информации. Например, NFKD("™") вернет обычную двухсимвольную строку "TM". Это может быть желательно для организации поиска и подобных операций, но каноническая декомпозиция для большинства ситуаций обычно дает более приемлемые результаты, чем совместимая декомпозиция.

Если не выполнить нормализацию самостоятельно, строка в программе необязательно будет иметь форму NFD или NFC; строки могут существовать в ненормализованном виде. Рассмотрим в качестве примера символ «ȭ», который является всего лишь строчной латинской буквой «о» с тильдой и знаком долготы над ней (в противоположность знаку долготы с тильдой над ним). Данная графема может быть представлена различным числом кодов, от одного до трех, в зависимости от формы: "\x{22D}" – в NFC, "\x{6F}\x{303}\x{304}" – в NFD, и "\x{F5}\x{304}" – без нормализации. В табл. 6.4 перечислено семь вариантов строчной латинской буквы «о» с тильдой и в некоторых случаях со знаком долготы.

Таблица 6.4. Каноническая головоломка

№ п/п	Начертание	NFC?	NFD?	Литерал	Кодовые пункты
1	õ	✓	–	"\x{F5}"	LATIN SMALL LETTER O WITH TILDE
2	õ	–	✓	"o\x{303}"	LATIN SMALL LETTER O, COMBINING TILDE
3	ȭ	✓	–	"\x{22D}"	LATIN SMALL LETTER O WITH TILDE AND MACRON
4	ȭ	–	–	"\x{F5}\x{304}"	LATIN SMALL LETTER O WITH TILDE, COMBINING MACRON
5	ȭ	–	✓	"o\x{303}\x{304}"	LATIN SMALL LETTER O, COMBINING TILDE, COMBINING MACRON
6	ō̃	–	✓	"o\x{304}\x{303}"	LATIN SMALL LETTER O, COMBINING MACRON, COMBINING TILDE
7	ō̃	✓	–	"\x{14D}\x{303}"	LATIN SMALL LETTER O WITH MACRON, COMBINING TILDE

В языке Perl за функции нормализации отвечает стандартный модуль Unicode::Normalize. **Как правило, весь текст Юникода, получаемый извне, желательно сначала пропускать через процедуру нормализации NFD, а весь текст Юникода, который выводится вовне, – через процедуру NFC, как показано ниже:**

```
use v5.14;
use strict;
use warnings;
use warnings FATAL => "utf8";  # возбуждать исключения,
                               # связанные с ошибками кодирования
use open qw(:std :utf8);

use Unicode::Normalize qw(NFD NFC);

while (my $line = <>) {
    $line = NFD($line);
    ...
} continue {
    print NFC($line);
}
```

Этот фрагмент читает текст в кодировке UTF-8 и автоматически декодирует его, возбуждая исключения при обнаружении ошибок кодирования. Первое, что делается в цикле – выполняется нормализация входной строки в форму канонической декомпозиции. Иными словами, все предварительно скомпонованные символы разбиваются на составляющие, к неописуемой радости редукционистов. При этом также переупорядочиваются все знаки, присоединяемые к разным частям базового кода[1].

Без нормализации вы не сможете даже подступиться к решению проблемы, связанной с комбинационными знаками. Рассмотрим графемы, представленные в табл. 6.4.

1 Если заумствовать, то следует сказать «согласно их каноническим комбинационным классам».

- Номер 4 – это ненормализованная графема. Такое иногда случается.
- Если предположить, что вы выполняете нормализацию NFD, графема 1 превратится в графему 2, 3 и 4 превратятся в графему 5, а графема 7 превратится в графему 6.
- Если предположить, что вы выполняете нормализацию NFC, графема 2 превратится в графему 1, 4 и 5 превратятся в графему 3, а графема 6 превратится в графему 7.
- Это означает, что путем нормализации в *любую* форму, NFD *или* NFC, можно обеспечить возможность сравнения графем 1–2, 3–5 и 6–7 посредством простого оператора eq.
- Однако, обратите внимание, что это три разных набора. ☹

Одна приятная новость состоит в том, что шаблоны Perl предлагают отличную поддержку графем, главное знать, как ею пользоваться. Метасимволу \X в регулярных выражениях соответствует единственный, видимый пользователем, символ, который на языке Юникода называется групповой графемой.[1]

Большинство групповых графем (но не все) состоит из базового кода и нуля или более комбинационных кодов. Одной широко распространенной двухсимвольной графемой, не имеющей комбинационных знаков, является "\r\n", которая обычно называется CRLF. Метасимвол \X соответствует комбинации CRLF как единой групповой графеме, потому что с точки зрения пользователя это единственный символ. В алфавите Japanese также имеются две расширенные графемы, не содержащие комбинационных знаков, HALFWIDTH KATAKANA VOICED SOUND MARK и HALFWIDTH KATAKANA SEMI-VOICED SOUND MARK.

Но обычно групповую графему можно представлять себе, как базовый символ (\p{Grapheme_Base}) с произвольным количеством комбинационных знаков, селекторов вариантов, японских огласовок, соединительных знаков нулевой ширины, а также несоединительных знаков (\p{Grapheme_Extend}*), следующих непосредственно за ним. Исключение составляет пара CRLF.

Фактически групповые графемы можно считать обычными графемами.[2]

Метасимвол \X в операциях поиска по шаблону соответствует любой из семи графем, представленных выше, и даже не требует приведения их к канонической форме. Хорошо, и что дальше? А вот с этого места начинаются сложности. Если вам потребуется знать о графеме больше, чем то, что она является графемой, вам придется проявить умеренную сообразительность при реализации поиска по шаблону. Приведение к NFD – условие *необходимое и достаточное*, чтобы:

- /^o/ сообщал, что все семь графем начинаются символом «o»;
- /^o\N{COMBINING TILDE}/ сообщал, что графемы 1–5 начинаются с символа «o» и тильды, но пропускал графемы 6 и 7;
- Найти все семь графем. Придется использовать /^o\pM*?\N{COMBINING TILDE}/.

---

[1] Строго говоря, метасимволу \X соответствует то, что в стандарте Юникода описывается, как расширенная групповая графема. Очевидно, разработчикам стандартов платят за количество слов.

[2] Именно это мы и делаем. Нам не платят за количество слов.

А вот попытка решить вопрос поиска полного символа, оставляющая за кадром такие тонкости, как использование \p{Grapheme_Extend} вместо \pM, а также \p{Grapheme_Base} (если уместно) вместо \PM:

```
$o_tilde_rx = qr{ o \pM *? \x{COMBINING TILDE} \pM* }x;
```

Гораздо более простой подход к сравнению строк (без учета диакритических знаков) приводится в следующем разделе, «Сравнение и сортировка строк Юникода».

Метасимвол \X – единственный в ядре Perl, знающий о существовании графем. Встроенные функции, такие как substr, length, index, rindex и pos манипулируют кодами символов, а не графемами. Поэтому \X – это ваш молоток. С ним весь набор Юникода начинает напоминать гвозди. Огромное количество гвоздей.

Представьте, что вам потребовалось обратить порядок следования кодов символов в строке «crème brûlée». Если предположить, что строка нормализована в форму NFD, вы получили бы в результате «éel̂urb em̀erc», тогда как в действительности требуется получить «eélûrb emèrc». Вместо этого следует с помощью \X извлечь список графем и переупорядочить их.

```
use v5.14;
use utf8;
my $cb = "crème brûlée";
my $bc = join("" => reverse($cb =~ /\X/g));
say $bc;    # "eélûrb emèrc"
```

Предположим, что переменная $cb ниже всегда содержит строку «crème brûlée», сравним операции над кодами и операции над графемами:

```
my $char_length = length($cb);      # 15 или 12
my $graph_count = 0;
$graph_count++ while $cb =~ /\X/g; # 12
```

Извлечь первую часть можно было бы, как показано ниже:

```
my $piece  = substr($cb, 0, 5),     # "crèm" или "crème"
my($piece) = $cb =~ /(\X{5})/;      # "crème"
```

А изменить последнюю часть следующим образом:

```
substr($cb, -6) = "fraîche";        # "crème brfraîche" или "crème fraîche"
$cb =~ s/\X{6}$/fraîche/;           # "crème fraîche
```

И вставить «bien»:

```
substr($cb, 5, 0) = " bien";   # "crèm biene brûlée" или "crème bien brûlée"
$cb =~ s/^\X{5}\K/ bien/       # "crème bien brûlée"
```

Обратите внимание, насколько нестабилен подход, основанный на кодах. В комментариях первый ответ соответствует строкам в NFD, а второй – строкам в NFC. Может показаться, что приведение строк к NFC решит все проблемы, но это не так. С одной стороны, графем, не имеющих предварительно скомпонованных знаков, бесконечно больше, чем графем, имеющих такие знаки, поэтому приведение к форме NFC не гарантирует избавление от комбинационных знаков.

Кроме того, форма NFC в действительности сложнее в использовании, поэтому мы рекомендуем всегда приводить входящие данные к форме NFD. Взгляните, как можно было бы выделить слова с двумя «e», такие как «crème» и «brûlée». Простейший и самый надежный способ:

```
/ e .*? e /x
```

будет работать только со строками, нормализованными в форму NFD, а не NFC. А если вы думаете, что, используя форму NFC, можно гарантировать то же самое, записав:

```
/ [eéè] .*? [eéè] /x
```

то быстро поймете, что ошиблись, столкнувшись со словом «crêpes». Добавление одного «ê» как будто решает проблему, но таким путем вы очень скоро придете к совершенно сумасшедшему шаблону:

```
/ [èéêëēĕėęȩẹě] [èéêëēĕėęȩẹě] /x   # два символа е в строке
```

который также окажется неработоспособным, если кто-то подсунет ему «е» с подчеркиванием, поскольку этот символ не имеет предварительно скомпонованной графемы. Если (и только если) ваши строки будут приведены к форме NFD, следующий шаблон будет работать всегда:

```
/ (?: (?=e) \X ){2} /x
```

Это решение обеспечивает надежный и неразрушающий способ сопоставления без учета акцентов: используйте метасимвол \X, соответствующий графеме, и введите ограничение, требующее, чтобы графема начиналась с искомого базового символа. Единственное, чего нельзя добиться таким способом – реализовать поиск в строках, приведенных к форме NFD (или NFKD), букв, которые не поддаются декомпозиции, потому что они считаются самостоятельными буквами.

Например, таким способом нельзя отыскать все символы «о» в слове «smørrebrød», потому что LATIN SMALL LETTER O WITH STROKE не имеет составного представления, где символ «о» был бы выделен из графемы. В имени «Ævar Arnfjörð Bjarmason» вы сможете отыскать все буквы «о» после декомпозиции, но не сможете отыскать символы «е» и «d», потому что LATIN CAPITAL LETTER AE не разбивается на «а» и «е», а LATIN SMALL LETTER ETH не превращается в «d».

Во всяком случае, при использовании декомпозиции. Однако сравнение с помощью специализированного объекта из Unicode::Collate позволило бы отыскать все три символа. В следующем разделе, «Сравнение и сортировка строк Юникода», мы покажем, как это делается.

Необходимость использовать метасимвол \X каждый раз, когда возникает потребность в использовании встроенных строковых функций, выглядит несколько странной. Альтернативное решение заключается в использовании модуля Unicode::GCString из CPAN.

Обычные строки в Perl всегда интерпретируются как последовательности кодов, но этот объектно-ориентированный модуль позволяет работать со строками, как с последовательностями групповых графем Юникода. Ниже показано, как можно использовать методы из этого модуля для выполнения обсуждавшихся выше операций над последовательностями графем:

```
my $gs = Unicode::GCString("crème brûlée");

say $gs->length();
say $gs->substr(0,5);
$gs->substr(-6, 6, "fraîche");
$gs->substr( 5, 0, " bien");
```

Теперь выбор формы нормализации не имеет значения, потому что метод length возвращает ответ в графемах, метод substr оперирует графемами, и можно даже использовать методы index и rindex для поиска литеральных подстрок, получая целочисленное смещение в графемах, а не в кодах символов.

Пожалуй, самый полезный метод в этом модуле – это метод columns. Представьте, что необходимо вывести несколько элементов меню, как показано ниже:

```
crème brûlée        £5.00
trifle              £4.00
toffee ice cream    £4.00
```

Как обеспечить выравнивание цен по вертикали? Даже если предположить, что вывод осуществляется с использованием моноширинного шрифта, следующее решение не поможет:

```
printf("%-25s £%.2f\n", $item, $price);
```

потому что Perl предполагает, что каждый код занимает ровно одну позицию в строке, что на практике не так.

Метод columns возвращает количество позиций, которые займет строка при выводе. Часто это число совпадает с количеством графем в строке, но не всегда. Некоторые символы Юникода считаются «широкими», в том смысле, что при выводе они занимают две позиции. Это настолько типично для символов восточноазиатских алфавитов, что в Юникоде существуют специальные свойства East_Asian_Width=Wide и East_Asian_Width=Full, которые указывают, что символ занимает две позиции при выводе.

Некоторые символы вообще не занимают позиции при выводе, и не только потому, что они являются комбинационными знаками: это могут быть символы управления или форматирования. Плюс ко всему к этому, некоторые комбинационные знаки занимают отдельную позицию при выводе. Единственное, на что вообще можно положиться при использовании моноширинного шрифта, это то, что размер каждого символа будет кратен ширине одной позиции.

Ниже демонстрируется одно из возможных решений дополнения строки до определенной длины:

```
sub pad {
    my($s, $width) = @_;
    my $gs = Unicode::GCString->new($s),
    return $gs . (" " x ($width - $gs->columns));
}

printf("%s £%.2f\n", pad($item, 25), $price);
```

Теперь ваши строки выравняются по вертикали, даже если будут содержать форматирующие символы, комбинационные знаки или широкие символы.

Несмотря на всю свою привлекательность, модуль Unicode::GCString в действительности является всего лишь вспомогательным модулем для другого, более крупного модуля, который решает более сложную проблему: модуля Unicode::LineBreak из CPAN. Последний реализует алгоритм разбиения строк (Unicode Line Breaking Algorithm), описываемый в UAX#14, приложении к стандарту Юникода. Он может пригодиться вам, когда потребуется разбить текст Юникода на абзацы, как это делает программа UNIX *fmt*(1) из модуля Text::Autoformat. В качестве примера

можно предложить программу *unifmt* из модуля Unicode::Tussle. Она все делает правильно, даже столкнувшись с восточноазиатскими широкими символами, табуляциями, комбинационными символами и невидимыми кодами форматирования.

Сравнение и сортировка строк Юникода

Встроенные функции sort и cmp не сравнивают строки по алфавитному признаку. Вместо этого сравниваются числовые значения кодов символов в одной строке с числовыми значениями кодов соответствующих символов в другой строке. Такой подход плохо работает с текстом, где наряду с символами, общими для разных языков, встречаются символы, характерные для каждого отдельно взятого языка. Это связано не только с наличием кодов, числовые значения которых не совпадают с алфавитным порядком, – числа и другие последовательности также могут вносить беспорядок, из-за того, что некоторые были добавлены в наборы символов, когда они были еще маленькими, а другие – когда наборы символов имели уже приличный размер, как Topsy. Например, символы показателей степеней 2 и 3 появились в Latin-1 на ранних этапах его становления. Поэтому, кстати, при сортировке они располагаются первыми:

```
use v5.14;
use utf8,
my @exes = qw( x⁷ x⁰ x⁸ x³ x⁶ x⁵ x⁴ x² x⁹ x¹ )
@exes = sort @exes;
say "@exes";

# выведет: x² x³ x¹ x⁰ x⁴ x⁵ x⁶ x⁷ x⁸ x⁹
```

Поскольку числовые значения кодов не соответствуют алфавитному порядку следования символов, строки будут сортироваться в порядке, который, не будучи совсем уж случайным, не соответствует ожиданиям пользователей. Функция sort хороша в основном для быстрого упорядочивания, когда порядок следования всегда будет одним и тем же, пусть и не совпадающим с алфавитным. Иногда этого вполне достаточно, но иногда...

Представляем стандартный модуль Unicode::Collate, который реализует алгоритм упорядочивания Юникода (Unicode Collation Algorithm, UCA), обладает широкими возможностями настройки и обеспечивает многоуровневую сортировку Юникода. Модуль обладает большим количеством замысловатых особенностей, но часто его применение ограничивается вызовом его метода sort без дополнительных параметров:

```
use v5.14;
use utf8;
use Unicode::Collate;
my @exes = qw( x⁷ x⁰ x⁸ x³ x⁶ x⁵ x⁴ x² x⁹ x¹ );
@exes = Unicode::Collate->new->sort(@exes);
say "@exes";

# выведет: x⁰ x¹ x² x³ x⁴ x⁵ x⁶ x⁷ x⁸ x⁹
```

По умолчанию модуль обеспечивает алфавитно-цифровую сортировку. В первом приближении, его принцип действия выглядит так: из текста удаляются все неалфавитноцифровые символы, после чего оставшееся сортируется без учета реги-

стра символов, не в числовом порядке кодов, а в порядке следования в алфавите. Такого рода сортировка используется в словарях, именно поэтому она иногда называется словарной, или лексикографической сортировкой.

Прежде чем люди привыкли пользоваться компьютерами, не умеющими правильно сортировать текст, именно такой порядок сортировки считался (и по сей день зачастую считается) верным. Название книги с запятой после первого слова в перечне названий должно располагаться рядом с таким же названием, не содержащим запятой. Запятые не должны оказывать влияния на сортировку, по крайней мере, когда этого не требуется. Запятые не являются естественной частью алфавитного порядка, как буквы и цифры.

Взгляните, какие результаты дает встроенная функция sort (которая сортирует строки так же, как аналогичная команда, имеющаяся в оболочке интерпретатора команд и в большинстве языков программирования):

```
% perl -e 'print sort <>' little-reds
Little Red Mushrooms
Little Red Riding Hood
Little Red Tent
Little Red, More Blue
Little, Red Rider
```

Что за ерунда? Слово «More» должно предшествовать словам «Mushrooms», слова «Rider» и «Riding» должны располагаться рядом, а слово «Tent» должно оказаться в конце. Даже с точки зрения ASCII это не алфавитный порядок; вот алфавитный порядок:

```
% perl -MUnicode::Collate -e
    print for Unicode::Collate->new->sort(<>)' little-reds
Little Red, More Blue
Little Red Mushrooms
Little, Red Rider
Little Red Riding Hood
Little Red Tent
```

Нам кажется, что вам понравятся результаты этой сортировки Юникода настолько, что вы захотите сохранить этот маленький сценарий на случай, когда потребуется сортировка обычного текста. Он предполагает, что исходный текст имеет кодировку UTF-8, и производит вывод тоже в кодировке UTF-8:

```
#!/usr/bin/perl
use warnings;
use open qw(:std :utf8);
use warnings qw(FATAL utf8);
use Unicode::Collate;
print for Unicode::Collate->new->sort(<>);
```

Более разносторонний вариант этого сценария доступен в виде программы *ucsort*, входящей в комплект Unicode::Tussle из архива CPAN.

Большинство людей считает, что с настройками по умолчанию модуль позволяет добиться вполне приемлемых результатов. Он уже знает, как сортировать буквы и числа, плюс умеет обращаться со всеми странностями Юникода, противоречащими ASCII-сортировке, как например упорядочение символов, которые согласно числовому порядку следования кодов оказываются далеко друг от друга, но

при сортировке должны оказываться рядом, учитывает все замысловатые правила Юникода, касающиеся регистра символов, принимает во внимание канонические эквиваленты строк и многое другое.

Плюс, если у вас имеются собственные предпочтения, модуль обладает практически неограниченным потенциалом настройки. Ниже демонстрируется несложная настройка, которая пригодится при сортировке названий книг и фильмов на английском языке. На этот раз символы верхнего регистра предшествуют символам нижнего регистра, перед сортировкой удаляются ведущие артикли и добавляются ведущие нули в числах, благодаря чему название «101 Dalmations» оказывается в списке ниже, чем «7 Brides for 7 Brothers».[1]

```
my $collator = Unicode::Collate->new(
    upper_before_lower => 1,
    preprocess => sub {
        local $_ = shift;
        s/^ (?: The | An? ) \n+ //x;        # отбросить артикли
        s/ ( \d+ ) / sprintf "%020d", $1 /xeg;
        return $_;
    },
);
```

Выше уже было показано, насколько более приемлемой выглядит алфавитная сортировка в сравнении с сортировкой по числовым значениям символов ASCII. В Юникоде ситуация обостряется. Даже если вы используете «всего лишь» английский язык, вам все равно придется сталкиваться не только с символами ASCII. Что если в ваших данных присутствует обозначение «10¢» или «£5»? Даже в тексте исключительно на английском языке могут встретиться фигурные кавычки, замысловатые дефисы и прочие специальные символы, отсутствующие в наборе ASCII. Даже если говорить исключительно о словах, которые можно найти в словаре английского языка, это не освобождает вас от необходимости предусматривать обработку особых случаев. Ниже приводится список слов из оксфордского словаря английского языка (Oxford English Dictionary), отсортированных (по колонкам) с использованием алгоритма UCA в режиме по умолчанию:

Allerød	fête	Niçoise	smørrebrød
après-ski	feuilleté	piñon	soirée
Bokmål	flügelhorn	plaçage	tapénade
brassière	Gödelian	prêt-à-porter	vicuña
caña	jalapeño	Provençal	vis-à-vis
crème	Madrileño	quinceañera	Zuñi
crêpe	Möbius	Ragnarök	α-ketoisovaleric acid
désœuvrement	Mohorovičić discontinuity	résumé	(α-)lipoic acid
Fabergé	moiré	Schrödinger	(β-)nornicotine
façade	naïve	Shijō	ψ-ionone

[1] Ведущие нули необходимы, потому что, несмотря на известность числовых значений цифровых кодовых пунктов, инструменты сортировки обычно не понимают, что число 9 должно предшествовать числу 10, если не использовать трюк, подобный этому.

Едва ли кто-то предпочел бы увидеть эти слова, отсортированные в порядке ASCII. Это не самое приятное зрелище. И это текст, содержащий только латинские символы. Взгляните на слова, встречающиеся в греческой мифологии, отсортированные по кодам символов:

Δύσις	Άσβολος	Διόνυσος	Φάντασος	Μεγαλήσιος
Ασβετος	Αγχίσης	Έσπερίς	Ἄγδιστις	Τελεσφόρος
Ασωπος	Λάχεσις	Ἕσπερος	Ἀστραῖος	Χρυσόθεμις
Θράσος	Νέμεσις	Εὔνοστος	Ασκληπιός	Ἀριστόδημος
Ιάσιος	Περσεύς	Ήφαιστος	Ἥφαιστος	Ἀριστόμαχος
Νέσσος	Άδραστος	Ηωσφόρος	Ἀρισταῖος	Λαιστρυγόνες
Πέρσης	Άλκηστις	Θρασκίας	Ἀσκαλαφος	
Πίστις	Αίγισθος	Πάσσαλος	Βορυσθενίς	
Χρύσος	Αργέστης	Πρόφασις	Ἑσπερίδες	

Даже если вы не знаете греческий алфавит, вы все равно сможете заметить, насколько сортировка по кодам не соответствует алфавитной: просто пробегите взглядом по первым буквам в каждом столбце. Видите, как они «прыгают» с места на место? При сортировке с использованием алгоритма UCA в режиме по умолчанию они располагаются правильном порядке:

Ἄγδιστις	Ασβετος	Ἑσπερίδες	Ιάσιος	Πίστις
Αγχίσης	Άσβολος	Έσπερίς	Λαιστρυγόνες	Πρόφασις
Άδραστος	Ἀσκάλαφος	Ἕσπερος	Λάχεσις	Τελεσφόρος
Αίγισθος	Ασκληπιός	Εὔνοστος	Μεγαλήσιος	Φάντασος
Άλκηστις	Ἀστραῖος	Ἥφαιστος	Νέμεσις	Χρυσόθεμις
Αργέστης	Ασωπός	Ήφαιστος	Νέσσος	Χρυσος
Ἀρισταῖος	Βορυσθενίς	Ηωσφόρος	Πάσσαλος	
Ἀριστόδημος	Διόνυσος	Θρασκίας	Περσεύς	
Ἀριστόμαχος	Δύσις	Θράσος	Πέρσης	

Нам удалось вас убедить? Теперь посмотрим, как в действительности работает алгоритм UCA, а затем познакомимся с его настройками.

Алгоритм Unicode Collation Algorithm предусматривает многоуровневую сортировку. Вы с ней уже сталкивались. Представьте теперь, что вы написали свою функцию сравнения для передачи в качестве функции обратного сравнения встроенной в язык `sort`, и ваша функция выглядит так:

```
@collated_text = sort {

    primary($a) <=> primary($b)
                ||
    secondary($a) <=> secondary($b)
                  ||
    tertiary($a) <=> tertiary($b)
                 ||
```

```
    quaternary($a) <=> quaternary($b)
} @random_text;
```

Это – многоуровневая сортировка. Если говорить упрощенно, именно так работает алгоритм UCA. Каждая из четырех функций возвращает число, определяющее вес данного уровня. Только когда на первом уровне обнаруживается совпадение, сравнение продолжается на втором уровне и т. д.

Ниже приводится несколько упрощенное, но достаточно полное описание алгоритма:

Первый уровень: сравнение букв

Проверяется равенство базовых букв.[1] На этом уровне игнорируются все символы, не являющиеся буквами – они просто пропускаются в процессе сканирования строки. Если буквы в одних и тех же относительных позициях не совпадают, это несоответствие определяет словарный порядок их следования.

Если выполняется сортировка текста, содержащего только символы из алфавита Latin, будет получен обычный алфавитный порядок «abc...», который вы изучали в школе, т.е. слово «Fred» будет предшествовать слову «freedom», как и словосочетание «free beer». Причина предшествования «free beer» слову «freedom» состоит в том, что пятая буква в первой строке – «b», а она предшествует пятой букве во второй строке – букве «d». Разобрались с механизмом работы? Это и есть словарный порядок. Пробелы не участвуют в сортировке.

Второй уровень: сравнение диакритических знаков

Если все буквы совпадают, на втором этапе выполняется сравнение диакритических знаков (точнее, комбинационных знаков: множества диакритических и комбинационных знаков перекрываются, но не полностью). По умолчанию сопоставление диакритических знаков выполняется слева направо, но направление можно поменять на обратное, как того требуют правила французского языка. (Классическим примером обычной сортировки по диакритическим знакам в направлении слева направо может служить последовательность слов `cote < coté < côte < côté`, которая во французском языке, согласно правилу сортировки справа налево, должна быть отсортирована иначе: `cote < côte < coté < côté`; два слова в середине меняются местами. Это связано с особенностями флективной морфологии французского языка.)

Третий уровень: сравнение регистров

Если все буквы и диакритические знаки совпадают, далее выполняется сравнение регистров символов. По умолчанию символы нижнего регистра предшествуют символам верхнего регистра, однако этот порядок можно легко изменить, добавив параметр `upper_before_lower => 1` при конструировании объекта, выполняющего сопоставление.

Четвертый уровень: сравнение всего остального

Если все буквы, диакритические знаки и регистры символов совпадают, производится сравнение всех остальных символов, таких как знаки пунктуации,

---

[1] А также цифр и некоторых других символов, которые вы, возможно, не считаете буквами (а они таковыми являются) – просто имейте в виду, что в данном случае под буквами подразумеваются не только буквы в прямом понимании.

специальные и пробельные символы, которые игнорировались на предыдущих уровнях. На этом уровне содержание строк учитывается полностью.

Использование уровней – дело добровольное. Вы можете, например, задействовать только первый уровень, где учитываются лишь базовые буквы и больше ничего.

Именно так выполняется сравнение строк «без учета акцентов» с использованием метода eq объекта, реализующего сравнение.

Нормализация не всегда действует наверняка. Например, нормализация бессильна, если вам требуется, чтобы буквы «o», «õ» и «ø» считались одинаковыми, потому что буква LATIN SMALL LETTER O WITH STROKE не поддается декомпозиции в нечто другое, начинающееся с базовой буквы «o». С другой стороны, при сравнении букв Unicode::Collate обычно считает «o», «õ» и «ø» одной и той же буквой. Но только не в алфавитах Swedish или Hungarian.

Аналогично обстоят дела с буквами «d» и «ð» – буква LATIN SMALL LETTER ETH не поддается декомпозиции в нечто другое, имеющее в составе базовую букву «d», но реализация алгоритма UCA считает их одной и той же буквой. Исключение составляет алфавит Icelandic (код «is» региональных настроек), где «d» и «ð» – совершенно разные буквы.

Если необходимо, чтобы объект, реализующий сравнение, игнорировал регистр, но учитывал акценты, укажите ему на необходимость выполнять сравнение только на первых двух уровнях и пропускать остальные, передав конструктору параметр level => 2.

Вот полный синтаксис всех необязательных параметров настройки конструктора для версии модуля v0.81:

```
$Collator = Unicode::Collate->new(
    UCA_Version => $UCA_Version,
    alternate => $alternate, # псевдоним для 'variable
    backwards => $levelNumber, # или \@levelNumbers
    entry => $element,
    hangul_terminator => $term_primary_weight
    ignoreName => qr/$ignoreName/,
    ignoreChar => qr/$ignoreChar/,
    ignore_level2 => $bool,
    katakana_before_hiragana => $bool,
    level => $collationLevel,
    normalization => $normalization_form,
    overrideCJK => \&overrideCJK,
    overrideHangul => \&overrideHangul,
    preprocess => \&preprocess,
    rearrange => \@charList,
    rewrite => \&rewrite,
    suppress => \@charList,
    table => $filename,
    undefName => qr/$undefName/,
    undefChar => qr/$undefChar/,
    upper_before_lower => $bool,
    variable => $variable,
)
```

Дополнительную информацию о параметрах конструктора можно найти на странице справочного руководства модуля. Хотя этот модуль и является частью стан-

дартной библиотеки Perl, он также доступен в архиве CPAN. Благодаря этому есть возможность обновлять его независимо от ядра Perl. Версия Perl v5.14 поставляется с модулем Unicode::Collate версии v0.73, поэтому совершенно очевидно, что с тех пор модуль обновился. Вам не нужно устанавливать самую современную версию Perl, чтобы использовать последнюю версию модуля. Он поддерживает даже такие старые версии Perl, как v5.6, и обеспечивает опережающую совместимость с последними версиями стандарта Юникода посредством аргумента конструктора UCA_Version.

Использование UCA с функцией sort

В реальной жизни встроенная функция sort обычно вызывается двумя способами: вообще без подпрограммы сравнения, либо с блоком кода в виде аргумента, играющим роль подпрограммы сравнения. В первом случае с успехом можно использовать метод sort из модуля Unicode::Collate, но во втором... Во втором случае можно воспользоваться другим методом объекта, реализующего сравнение, который называется getSortKey.

Предположим, что имеется программа, использующая встроенную функцию sort, как показано ниже:

```
@srecs = sort {
    $b->{AGE} <=> $a->{AGE}
              ||
    $a->{NAME} cmp $b->{NAME}
} @recs;
```

И вот нам пришло в голову, что текст следует отсортировать по полю NAME в алфавитном порядке, а не по числовым значениям кодов. Для этого достаточно просто попросить объект сравнения вернуть вам двоичный ключ сортировки для каждой текстовой строки, участвующей в сортировке. В отличие от обычного текста, если передать эти двоичные ключи оператору cmp, он волшебным образом отсортирует их в требуемом вам порядке.

Блок кода для передачи функции sort теперь выглядит так:

```
my $collator = Unicode::Collate->new();
for my $rec (@recs) {
    $rec->{NAME_key} = $collator->getSortKey( $rec->{NAME} );
}
@srecs = sort {
    $b->{AGE} <=> $a->{AGE}
              ||
    $a->{NAME_key} cmp $b->{NAME_key}
} @recs;
```

Конструктору можно передавать любые необязательные аргументы для достижения желаемых результатов, включая предварительную обработку.

Объекты, реализующие сравнение, можно также использовать для простого сравнения без учета акцентов и регистра символов. В этом есть определенный смысл – если есть возможность упорядочивать строки, значит, есть возможность определять их эквивалентность при определенных настройках упорядочивания. То есть, вам остается лишь выбрать нужную семантику упорядочивания. Например, если установить уровень сравнения 1, сравниваться будут только буквы, без учета ре-

гистра символов и наличия диакритических знаков. В этом вам помогут методы объекта, реализующего сравнение, такие как eq, substr и index. (Однако от нормализации придется отказаться, потому что иначе изменятся смещения кодов.) Например:

```
use v5.14;
use utf8;
use Unicode::Collate;
my $Collator = Unicode::Collate->new(
                  level => 1,
                  normalization => undef,
               );

my $full = "Gabriel García Márquez";
for my $sub (qw[MAR CIA]) {
    if (my($pos,$len) = $Collator->index($full, $sub)) {
        my $match = substr($full, $pos, $len);
        say "Соответствие литералу ‹$sub› найдено в ‹$full› в виде ‹$match›"
    }
}
```

Если запустить этот фрагмент, он выведет:

```
Соответствие литералу ‹MAR› найдено в ‹Gabriel García Márquez› в виде ‹Már›
Соответствие литералу ‹CIA› найдено в ‹Gabriel García Márquez› в виде ‹cía›
```

Пожалуйста, не сообщайте об этом в ЦРУ (CIA).

Сортировка с учетом региональных настроек

Алгоритм UCA с настройками по умолчанию с успехом справляется с текстами на английском и на многих других языках, включая ирландско-гэльский, индонезийский, итальянский, грузинский, голландский, португальский и немецкий (за исключением телефонных книг!). Однако для алфавитной (или не алфавитной – кому как вздумается) сортировки текстов на многих других языках требуется применить дополнительные настройки.

Например, в скандинавских языках буквы с диакритическими знаками при сортировке должны следовать за буквой «*z*», а не располагаться рядом с похожими. Даже в испанском языке есть свои особенности: буква «*ñ*» не считается обычной буквой «n» с тильдой, как буквы «*ã*» и «*õ*» в португальском языке. В испанском алфавите это самостоятельная буква (и называется она, конечно, *eñe*), которая должна следовать за буквой «*n*» и предшествовать букве «*o*». Из этого следует, что следующие слова должны сортироваться в таком порядке: *radio*, *ráfaga*, *ranúnculo*, *raña*, *rápido*, *rastrillo*. Обратите внимание, что слово *ranúnculo* должно идти перед словом *raña*, а не после него.

Учесть национальные особенности сортировки текста Юникода позволяет модуль Unicode::Collate::Locale. Он распространяется в составе Unicode::Collate и потому входит в состав Perl версии v5.14, а также устанавливается вместе с основным модулем при установке из CPAN.

Единственное отличие в API этих двух модулей заключается в наличии у конструктора из Unicode::Collate::Locale дополнительного параметра: регионального кода. На момент написания этих строк поддерживалось 70 различных регио-

нальных кодов, включая такие варианты, как немецкие телефонные книги (гласные с умляутами сортируются, как если бы они были обычными гласными, за которыми следует буква *«e»*), традиционный испанский (*«ch»* и *«ll»* считаются отдельными графемами со своим местоположением в алфавите), японский, и пять различных способов сортировки текста на китайском языке.

Пользоваться этим дополнительным параметром в действительности очень просто:

```
use Unicode::Collate::Locale;

$coll = Unicode::Collate::Locale->new(locale => "fr");

@french_text = $coll->sort(@french_text),
```

Поскольку Unicode::Collate::Locale является подклассом, наследующим Unicode::Collate, его конструктор принимает те же дополнительные аргументы, что и конструктор родительского класса, и объекты этого класса поддерживают те же методы, поэтому вы можете использовать их для организации поиска с учетом региональных особенностей, как было показано выше. Ниже демонстрируется пример выбора настроек сортировки, используемой в немецких телефонных книгах, где (к примеру) *«ae»* и *«ä»* считаются одной и той же буквой. Можно просто выполнить непосредственное сравнение

```
state $coll = new Unicode::Collate::Locale::
              locale => "de_ _phonebook".
         ;

if ($coll->eq($a, $b)) { ..  }
```

Или выполнить поиск:

```
use Unicode::Collate::Locale;
my $Collator = new Unicode::Collate::Locale:
                locale => "de_ _phonebook"
                level => 1,
                normalization => undef
             ;

my $full = "Ich muß Perl studieren
my $sub = "MUESS";
if (my ($pos,$len) = $Collator->index($full, $sub)) {
    my $match = substr($full, $pos, $len);
    say "Соответствие литералу ‹$sub› найдено в ‹$full› в виде ‹$match›"
}
```

Если запустить этот фрагмент, он выведет:

```
Соответствие литералу ‹MUESS› найдено в ‹Ich muß Perl studieren.› в виде ‹muß›
```

Дополнительные возможности

Всегда следует помнить, что такие сокращенные обозначения символьных классов в Perl, как \w, \s и даже \d, по умолчанию соответствуют многим символам Юникода, что диктуется определенными свойствами символов. Они перечислены в табл. 5.11 и отвечают формальным определениям из приложения «Annex C:

Compatibility Properties» к техническому стандарту Юникода «Unicode Technical Standard #18, Unicode Regular Expressions», версии 13, выпущенной в августе 2008.

Если вы привыкли в своих программах извлекать целые числа при помощи выражения (\d+), этот подход не всегда будет работать корректно с данными в Юникоде. По версии v6.0 стандарта Юникода, метасимволу \d соответствует 420 кодовых пунктов. Если вам это не подходит, используйте /\d/a или /(?a:\d)/, или задействуйте более конкретное свойство \p{POSIX_Digit}.

Если требуется извлекать любые десятичные цифры из любого алфавита и использовать их в программе как числа, можно воспользоваться функцией num из модуля Unicode::UCD.

```
use v5.14;
use utf8;
use Unicode::UCD qw(num);
my $num;
if ("४५६७" =~ /(\d+)/) {
    $num = num($1);
    printf "Найдено число: %d\n", $num;
    # Найдено число: 4567
}
```

Регулярные выражения позволяют проверять свойства символов, однако не способны определять, какими свойствами символ обладает (во всяком случае, без проверки всех свойство по списку). А иногда действительно бывает необходимо знать это. Например, если необходимо узнать, какому алфавиту принадлежит код символа, или к какой основной категории относится символ. Для этого можно использовать тот же самый модуль Unicode::UCD. Ниже приводится программа вывода свойств, которые могут пригодиться при поиске по шаблону.

```
use v5.14;
use utf8;
use warnings;

use Unicode::UCD       qw( charinfo );
use Unicode::Normalize qw( NFD );

## раскомментируйте следующую строку, чтобы использовать декомпозицию
my $mystery = ## NFD
              "७¾çΩ̃"
for my $chr (split //, $mystery) {
    my $ci = charinfo(ord $chr);
    print "U+", $$ci{code};
    printf  \N{%s}'."\n\t", $$ci{name};
    print " gc=",         $$ci{category},
    print " script=",     $$ci{script};
    print " BC=",         $$ci{bidi};
    print " mirrored=",   $$ci{mirrored},
    print " ccc=",        $$ci{combining};
    print " nv=",         $$ci{numeric};
    print "\n";
}
```

После запуска эта программа выведет:

```
U+096D \N{DEVANAGARI DIGIT SEVEN} gc=Nd script=Devanagari
       BC=L mirrored=L ccc=0 nv=7
U+00BE \N{VULGAR FRACTION THREE QUARTERS} gc=No script=Common A
       BC=ON mirrored=ON ccc=0 nv=3/4
U+00E7 \N{LATIN SMALL LETTER C WITH CEDILLA} gc=Ll script=Latin
       BC=L mirrored=L ccc=0 nv=
U+1F6F \N{GREEK CAPITAL LETTER OMEGA WITH DASIA AND PERISPOMENI}
       gc=Lu script=Greek BC=L mirrored=L ccc=0 nv=
```

Если удалить комментарий, препятствующий декомпозиции NFD, будет выведено:

```
U+096D \N{DEVANAGARI DIGIT SEVEN} gc=Nd script=Devanagari
       BC=L mirrored=L ccc=0 nv=7
U+00BE \N{VULGAR FRACTION THREE QUARTERS} gc=No script=Common
       BC=ON mirrored=ON ccc=0 nv=3/4
U+0063 \N{LATIN SMALL LETTER C} gc=Ll script=Latin
       BC=L mirrored=L ccc=0 nv=
U+0327 \N{COMBINING CEDILLA} gc=Mn script=Inherited
       BC=NSM mirrored=NSM ccc=202 nv=
U+03A9 \N{GREEK CAPITAL LETTER OMEGA} gc=Lu script=Greek
       BC=L mirrored=L ccc=0 nv=
U+0314 \N{COMBINING REVERSED COMMA ABOVE} gc=Mn script=Inherited
       BC=NSM mirrored=NSM ccc=230 nv=
U+0342 \N{COMBINING GREEK PERISPOMENI} gc=Mn script=Inherited
       BC=NSM mirrored=NSM ccc=230 nv=
```

Определение пользовательских границ в регулярных выражениях

Метасимволы \b и \B, обозначающие границу слова и не-(границу слова), соответственно, опираются на текущее определение метасимвола \w (здесь подразумевается, что они изменяют свое значение наряду с \w при переходе на семантику ASCII с помощью модификатора /a или /aa).

Если это не тот тип границ, что вы ищете, всегда можно создать собственное определение границ, опираясь на произвольные условия, такие как границы алфавита. Ниже приводится такое определение \b:

```
(?(?<=    \w) # если слева символ слова
      (?! \w) #     тогда справа должен быть не символ слова
  |   (?= \w) #     иначе справа должен быть символ слова
)
```

А так выглядит определение \B:

```
(?(?<=    \w) # если слева символ слова
      (?= \w) #     тогда справа должен быть символ слова
  |   (?! \w) #     иначе справа должен быть не символ слова
)
```

Теперь, когда вы знаете, как определяются границы слов и, соответственно, точки, таковыми не являющиеся, то сможете определять собственные границы, из-

меняя условия там, где находится метасимвол \w в шаблонах выше. Нужно лишь проследить за тем, чтобы ваше определение создавало условие фиксированной ширины – так его можно будет использовать в ретроспективной проверке. Это означает, что нельзя использовать такие метасимволы, как \X или \R, соответствия которым имеют переменную длину. Проще всего для этих целей использовать свойства или классы символов. Например, для определения символов из греческого алфавита можно было бы использовать свойство \p{Greek}, но лучше будет добавить свойство Inherited, чтобы не пропустить комбинационные знаки, поэтому используйте класс [\p{Greek}\p{Inherited}].

Например, ниже представлены подпрограммы для регулярного выражения, реализующие такого рода действия:

```
(?(DEFINE)
    (?<greeklish>          [\p{Greek}\p{Inherited}] )
    (?<ungreeklish>        [^\p{Greek}\p{Inherited}] )
    (?<greek_boundary>
        (?(?<=      (?&greeklish))
            (?!     (?&greeklish))
          | (?=     (?&greeklish))
        )
    )
    (?<greek_nonboundary>
        (?(?<=      (?&greeklish))
              (?=   (?&greeklish))
          |   (?!   (?&greeklish))
        )
    )
)
```

Для классов символов, являющихся результатом сложения, вычитания, отрицания и пересечения существующих свойств Юникода, как в подпрограмме <greeklish> выше, может пригодиться возможность определять пользовательские свойства. Нестандартные свойства выглядят как самые обычные свойства. Например:

```
sub IsGreeklish {
    return <<'END';
+utf8::IsGreek
+utf8::IsInherited
END
}
```

Теперь можно использовать \p{IsGreeklish} и \P{IsGreeklish} в шаблонах, скомпилированных в том же пакете, что и подпрограммы. Как собрать все это воедино, рассказывается в следующем разделе.

Укрепляем характер созданием символов

Чтобы определить собственное свойство, необходимо написать подпрограмму с именем свойства (см. главу 7). Из соображений безопасности (неквалифицированное) имя такой подпрограммы должно начинаться с префикса Is или In. Подпрограмма должна быть определена в пакете, где используется свойство (см. главу 10), т.е. если свойство требуется в нескольких пакетах, его необходимо либо

импортировать из модуля (см. главу 11), либо наследовать как метод класса из пакета, в котором оно определяется (см. главу 12).

Помимо соответствия организационным требованиям подпрограмма должна возвращать данные в формате файлов, находящихся в каталоге *PATH_TO_PERLLIB/unicode/Is*. То есть, возвращать список символов или диапазонов символов в шестнадцатеричном виде, по одному в строке. Если возвращается диапазон, два числа, представляющих диапазон, должны разделяться символом табуляции. Допустим, что требуется создать свойство, которое имело бы истинное значение для символов, попадающих в диапазон любой из японских слоговых азбук (kana), известных как хирагана (hiragana) и катакана (katakana). Для этого можно было бы определить два диапазона:

```
sub InKana {
    return <<'END ;
3040    309F
30A0    30FF
END
}
```

С другой стороны, это же свойство можно было бы определить через существующие свойства:

```
sub InKana {
    return <<'END';
+utf8::InHiragana
+utf8::InKatakana
END
}
```

Вычитание множеств выполняется с помощью префикса «–». Допустим, что необходимо обеспечить соответствие свойства только актуальным символам, а не диапазонам. Исключить неопределенные коды можно вот так:

```
sub IsKana {
    return <<'END ;
+utf8::InHiragana
+utf8::InKatakana
-utf8::IsCn
END
}
```

Можно также использовать дополнения наборов символов при помощи префикса «!»:

```
sub IsNotKana {
    return <<'END';
!utf8::InHiragana
-utf8::InKatakana
+utf8::IsCn
END
}
```

Пересечения определяются с помощью префикса «&», что бывает полезно для определения общих символов, входящих в два (или более) класса.

```
sub IsGraecoRomanTitle {<<'END_OF_SET'}
+utf8::IsLatin
+utf8::IsGreek
&utf8::IsTitle
END_OF_SET

sub IsGreekTitle {<<'END_OF_SET'}
+main::IsGraecoRomanTitle
-utf8::IsLatin
END_OF_SET
```

Важно помнить, что префикс «&» не может предшествовать первому множеству, иначе получится пересечение с пустым множеством, т.е. пустое множество.

В самом Perl используются точно такие же уловки для определения «традиционных» классов символов (таких как \w), когда вы включаете их в собственные классы символов (такие как [-.\w\s]). Кому то может показаться, что, чем сложнее правила, тем медленнее они будут выполняться. Но в действительности, как только Perl вычислит битовый шаблон для конкретного 64-битового образца свойства, он сохранит его, и никогда не будет выполнять повторный перерасчет шаблона. (Применение 64-битовых образцов даже не требует декодирования данных в кодировке UTF-8 при поиске.) Так что все классы символов, встроенные или ваши собственные, работают одинаково быстро.

Чтобы увидеть другой подход к настройке простым изменением синтаксиса классов символов в квадратных скобках, загляните в модуль Unicode::Regex::Set из архива CPAN.

Создавая пользовательские свойства с именами, назначенными пользователем, можно даже организовать управление неиспользуемыми кодами символов, не прибегая к загадочным числовым значениям. Например, в Юникод пока не был включен алфавит Tengwar (эльфийский)[1], хотя в планах он уже стоит – в конце концов, существует уже множество карт Средиземья. Но это не останавливает дизайнеров шрифтов, создающих великолепные шрифты для символов из алфавита Tengwar. Некоторые шрифты используют блоки кодов, зарезервированных непосредственно для алфавита Tengwar, но при этом большинство используемых кодов приходится на область для частного использования. В любом случае, этим кодовым числам еще не присвоены ни имена, ни свойства.

Для Perl это не является барьером, потому что он позволяет легко создавать собственные имена и свойства для символов. Один из существующих в Perl модулей поддержки алфавита Tengwar описывает имена символов следующим образом:

```
TENGWAR LETTER TINCO TENGWAR DIGIT ZERO
TENGWAR LETTER PARMA TENGWAR DIGIT ONE
TENGWAR LETTER CALMA TENGWAR DIGIT TWO
TENGWAR LETTER QUESSE TENGWAR DIGIT THREE
```

что позволяет пользоваться ими, как показано ниже:

```
if ($elvish =~ /\N{TENGWAR LETTER SILME NUQUERNA}/) {...}
```

---

1 Cirth (Кирт, Кертар) и Tengwar (Тенгвар) – изобретенные Дж. Р.Р. Толкиеном два вида эльфийских алфавитов – рунический (Кертар) и буквенный (Тенгвар). – *Прим. ред.*

без всяких помех. К кодам символов из алфавита Tengwar можно даже применять charnames::viacode, чтобы получать их имена. Более того, в модуле даже определены свойства символов из алфавита Tengwar:

```
In_Tengwar                 In_Tengwar_Numerals
In_Tengwar_Consonants      Is_Tengwar_Decimal
In_Tengwar_Vowels          Is_Tengwar_Duodecimal
In_Tengwar_Alphabetics     In_Tengwar_Marks
In_Tengwar_Punctuation     In_Tengwar_Alphanumerics
```

что позволяет писать такой программный код на Perl:

```
print "W" if /\p{In_Tengwar_Alphanumerics}/;
print "A" if /\p{In_Tengwar_Alphabetics}/;
print "C" if /\p{In_Tengwar_Consonants}/;
print "V" if /\p{In_Tengwar_Vowels}/;
```

или даже:

```
$TENGWAR_GRAPHEME = qr{
    (?>
        (?= \p{In_Tengwar} ) \P{In_Tengwar_Marks}
        \p{In_Tengwar_Marks} *
    ) | \p{In_Tengwar_Marks}
}x;
```

Попытки написать нечто подобное без применения имен абстракций для символов и свойств больше напоминают попытки писать программы с использованием числовых адресов в памяти вместо имен переменных. Безусловно, при большом желании это вполне возможно, но такой стиль не будет гармонировать с имеющимися возможностями, и подобные программы будет невероятно сложно читать и сопровождать. Позволяя определять собственный язык даже для таких узкоспециализированных применений, Perl помогает писать программный код, который выглядит чище и проще.

Ссылки

Язык Perl близко, насколько это возможно, придерживается стандарта Юникода во всех возможных аспектах. Этот стандарт включает различные приложения и технические отчеты. Некоторые из них имеют прямое отношение к темам, обсуждавшимся выше, включая:

UAX #44: Unicode Character Database

UTS #18: Unicode Regular Expressions

UAX #15: Unicode Normalization Forms

UTS #10: Unicode Collation Algorithm

UAX #29: Unicode Text Segmentation

UAX #14: Unicode Line Breaking Algorithm

UAX #11: East Asian Width

7

Подпрограммы

Как и многие другие языки, Perl позволяет программисту создавать собственные подпрограммы.[1] Эти подпрограммы могут определяться в любом месте основной программы, загружаться из других файлов с помощью ключевых слов do, require или use либо генерироваться на этапе выполнения с помощью eval. Их можно даже загружать на этапе выполнения программы – этот способ описан в разделе «Автозагрузка» в главе 10. Вызов подпрограмм может осуществляться косвенным образом, посредством переменной, содержащей имя подпрограммы или ссылку на нее, либо через объект, позволяя ему самому определить, какую подпрограмму вызвать. Можно также создавать анонимные подпрограммы, доступные только через ссылки, и использовать их для клонирования новых, почти идентичных функций через *замыкания* (*closures*), описываемые в одноименном разделе главы 8.

Синтаксис

Объявить именованную подпрограмму, не определяя ее, можно одной из следующих форм:

```
sub NAME
sub NAME PROTO
sub NAME       ATTRS
sub NAME PROTO ATTRS
```

Именованную подпрограмму можно объявить и определить, добавив блок *BLOCK*:

```
sub NAME             BLOCK
sub NAME PROTO       BLOCK
sub NAME       ATTRS BLOCK
sub NAME PROTO ATTRS BLOCK
```

---

[1] Еще мы называем их функциями, но в Perl функции и подпрограммы – одно и то же. Иногда мы будем называть их *методами*. Методы определяются так же, как подпрограммы, но вызываются иначе.

Чтобы создать анонимную подпрограмму или замыкание, достаточно опустить ее имя:

```
sub                 BLOCK
sub     PROTO       BLOCK
sub           ATTRS BLOCK
sub     PROTO ATTRS BLOCK
```

PROTO и *ATTRS* обозначают прототип и атрибуты; каждой из этих конструкций посвящен свой раздел этой главы. *PROTO* и *ATTRS* не столь важны: более существенными являются *NAME* и *BLOCK*, даже если они отсутствуют.

Но как вызвать подпрограмму без имени? Просто сохраните значение, возвращаемое формой sub, которая не только компилируется на этапе компиляции (очевидно), но и возвращает значение во время выполнения:

```
$subref = sub BLOCK;
```

Чтобы импортировать подпрограммы из другого модуля, нужно сказать:

```
use MODULE qw(NAME1 NAME2 NAME3...);
```

Прямой вызов подпрограмм осуществляется так:

```
NAME(LIST)    # & не обязателен, если есть скобки.
NAME LIST     # Скобки не обязательны при наличии предварительного объявления
              # sub или инструкции импортирования.
&NAME         # Передает подпрограмме текущее значение @_
              # (в обход прототипов).
```

Для косвенного вызова подпрограмм (по имени или по ссылке) используется синтаксис:

```
&$subref(LIST)  # & нельзя опускать при косвенном вызове
$subref->(LIST) # (если только не используется инфиксная запись).
&$subref        # Передает подпрограмме текущее значение @_
```

Официальное имя подпрограммы начинается с префикса &. Подпрограмму можно вызвать по имени с префиксом, но обычно он не используется, как и круглые скобки, если подпрограмма была предварительно объявлена. Однако префикс & является обязательным, когда нужно указать имя подпрограммы, чтобы, например, передать его в качестве аргумента функции defined или undef, либо создать ссылку на именованную подпрограмму, как в случае $subref = \&name. Префикс & обязательно должен использоваться и в косвенных вызовах подпрограмм, в конструкциях &$subref() или &{$subref}(). Однако более удобная конструкция $subref->() префикса & не требует. Дополнительные сведения о ссылках на подпрограммы приводятся в главе 8.

Perl не регламентирует использование заглавных букв в именах подпрограмм. Однако существует неформальное соглашение, в соответствии с которым имена функций, неявно вызываемых системой времени выполнения Perl (BEGIN, CHECK, UNITCHECK, INIT, END, AUTOLOAD, DESTROY и другие, перечисленные в главе 14), содержат только заглавные буквы, поэтому такого стиля именования желательно избегать. (Но имена подпрограмм, возвращающих константы, обычно тоже состоят только из заглавных букв. Это нормально. Надеемся...)

Семантика

Пока вы не слишком устали от всего этого синтаксиса, просто запомните, что обычное определение простой подпрограммы выглядит примерно так:

```
sub razzle {
    print "Ok, you've been razzled.\n";
}
```

а обычный способ ее вызова:

```
razzle();
```

Здесь мы опустили входные и выходные данные (аргументы и возвращаемые значения). Но модель обмена данными с подпрограммой, действующая в Perl, на самом деле очень проста: все аргументы передаются функции как один плоский список скаляров; множество возвращаемых значений также передается в точку вызова как один плоский список скаляров. Поскольку речь о списочном контексте, то, разумеется, все массивы и хеши, передаваемые таким образом, интерполируют свои элементы в плоский список, что сопряжено с потерей индивидуальности элементов (впрочем, эта проблема легко решается), а такая автоматическая интерполяция списка зачастую оказывается весьма удобной. Списки аргументов и возвращаемых значений могут содержать сколь угодно много или сколь угодно мало скалярных элементов (хотя можно наложить ограничения на список аргументов посредством прототипов). Это возможно благодаря тому, что архитектура Perl построена на *вариадических* (*variadic*) функциях, принимающих произвольное число аргументов, тогда как в язык C эти функции довольно неуклюже втиснуты, чтобы можно было вызывать *printf*(3).

Что ж, намереваясь создать язык, позволяющий передавать произвольное число произвольных аргументов, следует озадачиться тем, как облегчить обработку этих произвольных списков аргументов. Любые аргументы поступают в подпрограмму Perl в виде массива `@_`. Если функция вызвана с двумя аргументами, внутри функции они будут доступны как первые два элемента этого массива: `$_[0]` и `$_[1]`. Поскольку `@_` – это обычный массив с необычным именем, с ним можно делать все, что обычно делают с массивами.[1] Массив `@_` является локальным, но его элементы служат псевдонимами для реальных скалярных параметров. (Это называется семантикой передачи по ссылке.) Поэтому можно изменять фактические параметры, изменяя соответствующие элементы массива `@_`. (Однако это делается редко, потому что в Perl очень легко возвращать значения, представляющие интерес.)

По умолчанию подпрограмма (как и любой другой блок) возвращает значение последнего вычисленного выражения. Также можно явно использовать оператор `return`, чтобы определить возвращаемое значение и выйти из подпрограммы в любой ее точке. В любом случае, поскольку подпрограмма вызывается в скалярном или списочном контексте, то и последнее выражение в подпрограмме вычисляется в том же контексте.

---

[1] Это область, в которой Perl *более* ортогонален, чем типичные языки программирования.

Приемы работы со списками параметров

В Perl пока нет именованных формальных параметров, но на практике достаточно скопировать @_ в список значений my, прекрасно справляющийся с ролью списка формальных параметров. (Не случайно копирование значений изменяет семантику передачи по ссылке на семантику передачи по значению. Обычно именно такого поведения ожидают программисты, даже если не владеют специальной терминологией.) Вот типичный пример:

```
sub maysetenv {
    my($key, $value) = @_;
    $ENV{$key} = $value unless $ENV{$key};
}
```

Но вы не обязаны давать имена аргументам функции – и в этом весь смысл существования массива @_. Например, чтобы найти максимальное значение, можно напрямую перебрать элементы массива @_:

```
sub max {
    my $max = shift(@_);
    for my $item (@_) {
        $max = $item if $max < $item;
    }
    return $max;
}

$bestday = max($mon,$tue,$wed,$thu,$fri);
```

Аргументы со строго определенным порядком прекрасно подходят для функций с небольшим количеством параметров, но по мере увеличения их числа становится сложно запоминать, какой аргумент для чего используется, какие являются необязательными, а у каких имеются значения по умолчанию. Более гибкое решение этой проблемы: передавать подпрограмме аргументы в виде пар имя/значение. Первый элемент в каждой паре – имя аргумента; второй – его значение. Это помогает создавать самодокументирующий код, ведь по именам параметров можно понять, для чего они предназначены, не заглядывая в полное определение функции. Более того, пользователям вашей функции уже не нужно запоминать порядок следования аргументов, а неиспользуемые аргументы они вольны опускать. Мы настоятельно рекомендуем стиль, основанный на именованных параметрах.

Фокус в том, чтобы присвоить список аргументов @_ хешу:

```
configuration(PASSWORD => "xyzzy", VERBOSE => 9, SCORE => 0);

sub configuration {
    my %options = @_;
    print "Максимально подробно.\n" if $options{VERBOSE} == 9;
}
```

Это очень гибкий подход, и, чтобы это продемонстрировать, мы приведем пример из рецепта «Передача именованных параметров» (книга «Perl Cookbook», раздел «Подпрограммы»)[1].

---

[1] Т. Кристиансен, Н. Торкингтон «Perl: библиотека программиста». – Пер. с англ. – СПб.: Питер, 2001.

```
thefunc(INCREMENT => "20s", START => "+5m", FINISH => "+30m");
thefunc(START => "+5m", FINISH => "+30m");
thefunc(FINISH => "+30m");
thefunc(START => "+5m", INCREMENT => "15s");
```

В то же время, в подпрограмме создается хеш, заполняемый значениями по умолчанию, а также значениями из массива с парами именованных параметров.

```
sub thefunc {
    my %args = (
        INCREMENT => "10s",
        FINISH    => 0,
        START     => 0,
        @_,         # фактические аргументы затирают установки по умолчанию
    );
    if ($args{INCREMENT} =~ /m$/ ) { ... }
    ...
}
```

Именование аргументов с последующим присваиванием @_ хешу %args позволяет избавиться от необходимости запоминать порядок следования аргументов и делает любой аргумент необязательным, автоматически назначая ему значение по умолчанию.

С другой стороны, есть ситуации, когда мы предпочтем неименованные аргументы. Например, если необходимо изменить фактические аргументы функции:

```
upcase_in($v1, $v2);      # эта подпрограмма изменяет $v1 и $v2
sub upcase_in {
    for (@_) { $_ = uc($_) }
}
```

Конечно, таким способом нельзя изменять константы. Окажись аргумент скалярным литералом, например строкой "хоббит", или скалярной переменной, доступной только для чтения, например $1, при попытке изменить его Perl возбудил бы исключение (предположительно, фатальное; возможно, угрожающее вашей карьере). Например, следующий вызов не будет работать:

```
upcase_in("фредерик");
```

Было бы безопаснее, если бы функция upcase_in возвращала копии своих параметров, а не изменяла их по месту:

```
($v3, $v4) = upcase($v1, $v2);
sub upcase {
    my @parms = map { uc } @_;
    # Вызов выполнен в списочном контексте?
    return wantarray ? @parms : $parms[0];
}
```

Обратите внимание, что этой функции (не имеющей прототипа) безразлично, что ей передается: скаляры или массивы. Perl расплющит все в один большой, длинный, плоский список аргументов @_. Это один из случаев, когда проявляется простота передачи аргументов в Perl. Функция upcase будет отлично работать и без изменения определения upcase, даже если передать ей такие аргументы:

```
@newlist = upcase(@list1, @list2);
@newlist = upcase( split /:/, $var )
```

Но не поддавайтесь соблазну написать так:

```
(@a, @b) = upcase(@list1, @list2); # НЕВЕРНО
```

Почему? Потому что подобно плоскому списку входных параметров @_, список возвращаемых значений является плоским. В результате все значения окажутся в @a, а в @b запишется пустой список. Как этого избежать, рассказывается в разделе «Передача ссылок».

Индикация ошибок

Если потребуется организовать возврат из функции так, чтобы в месте вызова можно было определить, что произошла ошибка, проще всего это сделать с помощью простой команды return без аргумента. В результате при вызове функции в скалярном контексте возвращается undef, а в списочном – пустой список.

В крайнем случае для индикации ошибки можно возбудить исключение. Однако пользуйтесь этим средством экономно, иначе вся программа окажется замусоренной обработчиками исключений. Например, неудачная попытка открыть файл едва ли является исключительным событием для функции открытия файла. А вот если проигнорировать эту ошибку, мы вполне можем столкнуться с таким событием. Встроенная функция wantarray возвращает undef, если ваша функция вызвана в пустом контексте, и с ее помощью можно узнать, что вас игнорируют:

```
if ($something_went_awry) {
    return if defined wantarray; # хорошо, не пустой контекст.
    die "Посмотри-ка на мою ошибку, бестолочь!!!\n";
}
```

Вопросы областей видимости

Подпрограммы могут вызываться рекурсивно, потому что каждый вызов получает собственный массив аргументов, даже если подпрограмма вызывает саму себя. Если подпрограмма вызывается по имени с префиксом &, список аргументов можно не передавать. На этот случай предусмотрена особая модель поведения: вызываемой подпрограмме неявно передается массив @_ вызывающей подпрограммы. Это механизм повышения эффективности, и новичкам, возможно, стоит его избегать.

```
&foo(1,2,3),  # передача трех аргументов
foo(1,2,3);   # то же самое

foo(),        # передача пустого списка
&foo();       # то же самое

&foo;         # foo() получит текущие аргументы, как foo(@_), но быстрее!
foo;          # то же, что и foo(), если было объявление sub foo,
              # иначе голое слово "foo"
```

Форма вызова префиксом & не только делает необязательным список аргументов, но и отключает проверку прототипа для передаваемых аргументов. Отчасти это обусловлено историческими причинами, а отчасти обеспечивает удобную возмож-

ность мошенничать тем, кто хорошо понимает, что делает. См. раздел «Прототипы» далее в этой главе.

Переменные, используемые в функции, но не объявленные в ней, не обязательно будут глобальными переменными: они подчиняются обычным правилам Perl для областей видимости блоков. Как рассказывалось в разделе «Имена» главы 2, поиск имен переменных сначала выполняется в окружающей лексической области (или областях) видимости, а затем в области видимости пакета. Поэтому подпрограмме доступны все переменные my или state из охватывающей области видимости.

Например, следующая функция bumpx использует лексическую переменную $x, областью видимости которой является файл, поскольку область видимости, содержащая объявление my (т.е. сам файл), не была закрыта до определения подпрограммы:

```
# начало файла
my $x = 10;         # объявление и инициализация переменной
sub bumpx { $x++ } # функция может видеть внешнюю лексическую переменную
```

Программисты на C и C++, вероятно, сочтут $x «файловой статической» переменной. Она недоступна функциям в других файлах, но глобальна с точки зрения функций, определенных после объявления my. Программисты на C, ищущие в Perl «статические переменные» для файлов или функций, не найдут в нем ключевого слова «static». Программисты на Perl обычно избегают этого слова, потому что статические системы мертвы и скучны, да и вообще исторически сложилось так, что это слово затаскали.

В лексиконе Perl нет слова «static», но программисты на Perl легко создают переменные, локальные для функций и сохраняющие свое значение между вызовами. Для этого служит похожее понятие state-переменных, и мы обсудим его чуть позже. Но это не единственный способ получить такое поведение. Более мощные примитивы областей видимости Perl сочетаются с автоматическим управлением памятью так, как и не снилось ищущим ключевое слово «static».

Лексические переменные не уничтожаются автоматически при выходе из их области видимости; они попадают в переработку, только когда больше *не используются*, что значительно важнее. Чтобы создать локальную переменную, сохраняющую свое значение между вызовами функции, заключите всю функцию в дополнительный блок и поместите туда же объявление my. В блок можно поместить сразу несколько функций, и все они получат доступ к переменной, которая иначе была бы недоступна:

```
{
    my $counter = 0;
    sub next_counter { return ++$counter }
    sub prev_counter { return --$counter }
}
```

Как всегда, лексическая переменная доступна только коду в той же лексической области видимости. Но имена функций в этом примере доступны глобально (внутри пакета), и поскольку функции определены внутри области видимости $counter, они могут обращаться к этой переменной, недоступной для других функций.

Если функция загружена через require или use, проблем, скорее всего, не возникнет. Если же все происходит в основной программе, нужно, чтобы на этапе выпол-

нения присваивание переменной my выполнялось достаточно рано. Для этого можно поместить весь блок перед основной программой, либо заключить его в блок BEGIN или INIT, чтобы гарантировать исполнение до запуска программы:

```
BEGIN {
    my @scale = ("A" .. "G");
    my $note = -1,
    sub next_pitch { return $scale[ ($note += 1) %= @scale ] };
}
```

Блок BEGIN не влияет на определение подпрограммы и на сохранность значений лексических переменных, используемых подпрограммой. Он нужен, только чтобы гарантировать инициализацию переменных до первого вызова подпрограммы. Объявлениям локальных и глобальных переменных посвящены разделы «my», «state» и «our» главы 27. Конструкции BEGIN и INIT описываются в главе 16.

Чтобы упростить объявление переменных как можно ближе к месту их использования, в язык Perl было введено объявление state, представляющее собой разновидность объявления my. Чтобы использовать его, включите в код прагму, регламентирующую версию Perl (не ниже v5.10).

После этого ключевое слово state можно будет использовать для объявления лексических переменных, инициализируемых только один раз:

```
use v5.14,

sub bumpx {
    state $x = 10;      # инициализируется только один раз
    return $x++;
}
```

Эта функция действует в точности, как предыдущая, возвращая сначала 10, затем 11, затем 12 и т.д. А следующая функция создает локальный статический хеш, хранящий счетчики различных значений, полученных функцией:

```
sub seen_count {
    state %count;
    my $item = shift(),
    return ++$count{$item}
}
```

Декларация переменной state отличается от прочих деклараций переменных тем, что позволяет инициализировать только простую скалярную переменную. Массивы и хеши тоже можно объявлять как state-переменные, но их нельзя инициализировать тем же волшебным образом, что скаляры. Фактически, это не является таким уж большим ограничением, потому что сохраняется возможность инициализировать ссылку на желаемый тип, так как ссылка – это скаляр. Например, вместо:

```
# нельзя использовать state %hash = (....)
my %hash = (
    READY   => 1,
    WILLING => 1,
    ABLE    => 1,
);
```

хеш можно объявить как state-переменную иначе:

```
state $hashref = {
    READY   => 1,
    WILLING => 1,
    ABLE    => 1,
};
```

Ниже показано, как реализовать функцию next_pitch, описанную выше, с использованием state-переменных:

```
sub next_pitch {
    state $scale = ["A" .. "G"];
    state $note  = -1;
    return $scale->[ ($note += 1) %= @$scale ];
}
```

Основное преимущество объявления state заключается в отсутствии необходимости использовать блок BEGIN (или UNITCHECK), чтобы гарантировать инициализацию переменных до вызова функции.

Наконец, когда мы говорим, что state-переменные инициализируются только один раз, мы не имеем в виду, что state-переменные являются общими в отдельных замыканиях. Напротив, каждая из таких переменных инициализируется отдельно. Этим state-переменные отличаются от static-переменных в других языках.

Например, в обеих версиях приведенного ниже программного кода переменная $epoch является лексической переменной, локальной для возвращаемого функцией замыкания. При этом в функции timer_then она инициализируется до того, как функция вернет замыкание, а в timer_now инициализация переменной $epoch откладывается до момента, когда функция вернет замыкание в первый раз:

```
sub timer_then {
    my $epoch = time();
    return sub {
        ...
    };
}

sub timer_now {
    return sub {
        state $epoch = time();
        ...
    };
}
```

Передача ссылок

Если потребуется передать в функцию или из нее более одного массива с сохранением их структуры, используйте механизм явной передачи по ссылке. Но прежде следует разобраться в особенностях работы ссылок, описанных в главе 8. В противном случае этот раздел может показаться вам весьма загадочным. Впрочем, всегда ведь можно поразглядывать картинки...

Вот несколько простых примеров. Сначала определим функцию, ожидающую получить ссылку на массив. Если массив большой, передача ссылки выполняется значительно быстрее, чем передача длинного списка значений:

```
$total = sum ( \@a );

sub sum {
    my ($aref)  = @_;
    my ($total) = 0;
    for (@$aref) { $total += $_ }
    return $total;
}
```

Передадим функции несколько массивов, и заставим ее извлечь последний элемент каждого из них, вернув новый список, состоящий из этих бывших последних значений:

```
@tailings = popmany ( \@a, \@b, \@c, \@d );

sub popmany {
    my @retlist = ();
    for my $aref (@_) {
        push @retlist, pop @$aref;
    }
    return @retlist;
}
```

Следующая функция находит пересечение множеств и возвращает список ключей, имеющихся во всех переданных хешах:

```
@common = inter( \%foo, \%bar, \%joe );
sub inter {
    my %seen;
    for my $href (@_) {
        while (my $k = each %$href ) {
            $seen{$k}++;
        }
    }
    return grep { $seen{$_} == @_ } keys %seen;
}
```

До сих пор мы применяли самый обычный механизм для возврата списка. А что если потребуется передать или вернуть хеш? Если хеш только один или вы не возражаете, чтобы несколько хешей объединились в один, сойдет обычное соглашение по вызовам, хотя обходится оно дороговато.

Как уже отмечалось, беда подкарауливает при такой записи:

```
(@a, @b) = func(@c, @d);
```

или такой:

```
(%a, %b) = func(%c, %d);
```

Такая запись попросту не работает. Она всего лишь устанавливает `@a` или `%a` и очищает `@b` или `%b`. Кроме того, вызываемая функция не получит в качестве аргументов два отдельных массива или хеша: она получит один длинный список в `@_`.

Допустим, требуется, чтобы функция использовала ссылки и для входных, и для выходных данных. Следующая функция принимает в качестве аргументов две ссылки на массивы и возвращает две ссылки на массивы, упорядоченные по числу элементов в этих массивах:

```
($aref, $bref) = func(\@c, \@d);
print "@$aref больше, чем @$bref\n";
sub func {
    my ($cref, $dref) = @_;
    if (@$cref > @$dref) {
        return ($cref, $dref);
    } else {
        return ($dref, $cref);
    }
}
```

О передаче дескрипторов файлов или каталогов в функции и из них рассказывается в разделах «Ссылки на дескрипторы файлов» и «Ссылки на таблицы символов» главы 8.

Прототипы

Perl позволяет программисту определять собственные функции и вызывать их так же, как встроенные функции Perl. Рассмотрим push(@array, $item), ожидающую получить ссылку на @array, а не список значений в @array, чтобы можно было модифицировать массив. *Прототипы* позволяют объявлять подпрограммы, принимающие аргументы подобно многим встроенным функциям, т.е. с определенными ограничениями на число и типы аргументов. Мы называем их «прототипы», но они больше похожи на автоматические шаблоны контекста вызова, чем на то, о чем могут подумать программисты на C или Java. С помощью этих шаблонов Perl автоматически добавляет неявные символы обратной косой черты, вызовы scalar и все прочее, благодаря чему можно получить соответствие шаблону. Например, если объявить:

```
sub mypush (+@);
```

функция mypush будет принимать аргументы в точности, как это делает push. Чтобы все получилось, объявление функции должно быть видимо на этапе компиляции. Прототип влияет на интерпретацию вызовов функции, только когда опущен символ &. Иными словами, если вызвать ее как встроенную функцию, она будет вести себя как встроенная функция. А если вызвать ее как обычную подпрограмму, она будет вести как обычная подпрограмма. Символ & подавляет проверку прототипа и соответствующие контекстно-зависимые эффекты.

Прототипы принимаются во внимание только на этапе компиляции, и отсюда естественным образом следует, что они не влияют на ссылки на подпрограммы вида \&foo или косвенные вызовы подпрограмм, такие как &{$subref} или $subref->(). На вызовы методов прототипы также не оказывают влияния. Дело в том, что фактически вызываемая функция не определена на этапе компиляции, но зависит от иерархии наследования, которая устанавливается в Perl динамически.

Основное назначение прототипов – дать программисту возможность определять подпрограммы, действующие как встроенные функции. В табл. 7.1 перечислено

несколько прототипов, которые можно использовать для эмуляции соответствующих им встроенных функций.

Таблица 7.1. Прототипы для эмуляции встроенных функций

Объявление	Вызов
`sub mylink ($$)`	`mylink $old, $new`
`sub myreverse (@)`	`myreverse $a,$b,$c`
`sub myjoin ($@)`	`myjoin ":",$a,$b,$c`
`sub mypop (;+)`	`mypop @array`
`sub mysplice (+;$$@)`	`mysplice @array,@array,0,@pusnme`
`sub mykeys (+)`	`mykeys %{$hashref}`
`sub mypipe (**)`	`mypipe READHANDLE, WRITEHANDLE`
`sub myindex ($$;$)`	`myindex &getstring, "substr`
	`myindex &getstring, "substr", $start`
`sub mysyswrite (-$;$$)`	`mysyswrite OUTF, $buf`
	`mysyswrite OUTF, $buf, length($buf)-$off, $off`
`sub myopen (*;$@)`	`myopen HANDLE`
	`myopen HANDLE, $name`
	`myopen HANDLE, "-|", @cmd`
`sub mysin (_)`	`mysin $a`
	`mysin`
`sub mygrep (&@)`	`mygrep { /foo/ } $a,$b,$c`
`sub myrand ($)`	`myrand 42`
`sub mytime ()`	`mytime`

Каждый символ прототипа (они перечислены в скобках в левой колонке), предваренный обратной косой чертой, представляет фактический аргумент (пример которого приведен в правой колонке), который обязательно должен начинаться с этого символа. Как первый аргумент `keys` должен начинаться символом `%` или `$`, так и первый аргумент `mykeys` должен начинаться символом `%`. Эту задачу решает специальный прототип `+`, который является сокращением для `\[@%]`.[1]

Чтобы определить несколько допустимых типов аргумента с обратной косой чертой, можно использовать групповую форму записи `\[]`. Например, объявление:

```
sub myref (\[$@%&*])
```

позволяет вызывать `myref` любым из следующих способов, а Perl сам позаботится, чтобы функция получила ссылку на указанный аргумент:

```
myref $var
myref @array
myref %hash
myref &sub
myref *glob
```

---

[1] Прототип операторов хешей несколько раз менялся. В v5.8 он выглядел как `\%`, в v5.12 – как `\[@%]`, а в v5.14 он превратился в просто `+`.

Точка с запятой отделяет обязательные аргументы от необязательных. (Перед @ или % ее употребление излишне, так как списки могут быть пустыми.) Символы прототипа без обратной косой черты имеют особое значение. Каждый символ @ или % без обратной косой черты поглощает все остальные фактические аргументы и устанавливает списочный контекст. (Это эквивалентно *LIST* в описании синтаксиса.) Для аргумента, представленного символом $, устанавливается скалярный контекст. Символ & требует ссылки на именованную или анонимную подпрограмму.

В качестве последнего символа прототипа или непосредственно перед точкой с запятой вместо $ можно использовать _. Если этот аргумент не будет передан, вместо него будет использоваться текущая переменная $_. Например:

```
sub mymkdir(_;$) {
    my $dirname = shift();
    my $mask = @_ ? shift() . 0777;
    my $mode = $mask &~ umask();
    ...
}

mymkdir($path, 01750);
mymkdir($path);
mymkdir();    # будет передано значение $_
```

Прототип + является особой альтернативой символу $, действующей подобно \[@%], когда передается литерал массива или переменная-хеш, в остальных случаях он принудительно устанавливает скалярный контекст для аргумента. Это удобно для таких функций, которые могут принимать не только литерал массива (или хеша), но и ссылку на него:

```
sub mypush (+@) {
    my $aref = shift,
    die "Не массив и не ссылка на массив " unless ref($aref) eq "ARRAY";
    push @$aref, @_;
}
```

При использовании прототипа + функция всегда должна проверять допустимость типа аргумента. (Мы преднамеренно реализовали его так, чтобы он не работал с объектами, ведь в противном случае поощрялась бы возможность нарушения инкапсуляции объектов.)

Символ * позволяет подпрограмме принимать в соответствующей позиции все, что было бы принято встроенной функцией как дескриптор файла: голое имя, константу, скалярное выражение, typeglob или ссылку на typeglob. Это значение будет доступно в подпрограмме как простой скаляр или (в двух последних случаях) как ссылка на typeglob. Чтобы такие аргументы всегда преобразовывались в ссылки на typeglob, используйте Symbol::qualify_to_ref:

```
use Symbol 'qualify_to_ref';

sub myfileno (*) {
    my $fh = qualify_to_ref(shift, caller);
    ...
}
```

Обратите внимание, что последние три прототипа в табл. 7.1 обрабатываются особым образом. mygrep интерпретируется как настоящий списочный оператор,

myrand – как настоящий унарный оператор с тем же приоритетом, что и rand, а mytime – как настоящая функция без аргументов, типа time.

Это значит, что выражение:

```
mytime +2;
```

дает в результате mytime() + 2, а не mytime(2), как было бы при разборе без прототипа или с унарным прототипом.

Пример mygrep показывает также, что & обрабатывается особым образом в качестве первого аргумента. Обычно прототип & требует наличия аргумента вроде \&foo или sub{}. Однако если это первый аргумент, можно пропустить sub в объявлении анонимной подпрограммы и просто передать голый блок в позиции «косвенного объекта» (без последующей запятой). Изящество прототипа & состоит в том, что он позволяет создавать новый синтаксис при условии, что & находится в первой позиции:

```
sub try (&$) {
    my ($try, $catch) = @_;
    eval { &$try };
    if ($@) {
        local $_ = $@;
        &$catch;
    }
}
sub catch (&) { $_[0] }

try {
    die "phooey";
} # не конец вызова функции!
catch {
    /phooey/ and print "unphooey\n";
};
```

Этот пример выведет "unphooey". Здесь try вызывается с двумя аргументами: анонимной функцией {die "phooey";} и значением, возвращаемым функцией catch, которым в данном случае оказывается ее первый аргумент – весь блок другой анонимной функции. Функция try вызывает первый аргумент-функцию в блоке eval, который перехватит ошибку, если она возникнет. В случае ошибки вызывается вторая функция с локальной версией глобальной переменной $_, куда предварительно сохраняется возбужденное исключение.[1] Если все это выглядит для вас полной тарабарщиной, почитайте о функциях die и eval в главе 27, а затем об анонимных функциях и замыканиях в главе 8. Если же вас это заинтересовало, обратите внимание на модуль Try::Tiny из CPAN, где такая технология применяется для реализации структурированной обработки исключительных ситуаций с использованием предложений try, catch и finally.

Вот новая реализация оператора grep BLOCK[2] (встроенная функция, конечно, более эффективна):

```
sub mygrep (&@) {
    my $coderef = shift;
```

---

1 Да, остались нерешенными проблемы, связанные с областью видимости @_. Но мы пока не будем затрагивать этот вопрос.

2 Его невозможно реализовать в форме grep *EXPR*.

```
        my @result;
        for my $_ (@_) {
            push(@result, $_) if &$coderef;
        }
        return @result;
    }
```

Кое-кто предпочел бы иметь полные буквенно-цифровые прототипы. Буквы и цифры намеренно выведены из прототипов с целью когда-нибудь ввести формальные именованные параметры. (Поживем – увидим.) Основная задача имеющегося в настоящее время механизма состоит в том, чтобы дать создателям модулей возможность до определенной степени проверять пользователей этих модулей.

Встроенная функция prototype позволяет извлекать прототипы пользовательских и встроенных функций; подробности читайте в главе 27. Чтобы изменить прототип функции «на лету», используйте функцию set_prototype из стандартного модуля Scalar::Util. Например, если потребуется, чтобы функции NFD и NFC из модуля Unicode::Normalize действовали подобно прототипу «_», можно выполнить следующие операции:

```
use Unicode::Normalize qw(NFD NFC);

BEGIN {
    use Scalar::Util "set_prototype";
    set_prototype(\&NFD => "_");
    set_prototype(\&NFC => "_");
}
```

Подставляемые функции-константы

Функции с пустым прототипом (), т.е. не принимающие никаких аргументов, интерпретируются аналогично встроенной функции time. Что интересно, компилятор считает такие функции кандидатами на подстановку (inlining). Если после оптимизации и свертки констант результатом такой функции является константа или скаляр с лексической областью видимости без других ссылок, это значение будет использоваться вместо вызова данной функции. Однако вызовы вида *&NAME* никогда не преобразуются в подстановку и не подвержены другим эффектам, связанным с прототипами. (Читайте описание прагмы constant в главе 29, если вас интересует простой способ объявлять такие константы.)

Обе следующие версии функций для вычисления числа π компилятор сделает подставляемыми:

```
sub pi () { 3.14159 }         # Не точное значение, но близко к тому
sub PI () { 4 * atan2(1, 1) } # Точно, насколько возможно
```

Подстановка будет выполнена и для всех следующих функций, поскольку Perl может определить возвращаемые этими функциями значения на этапе компиляции:

```
sub FLAG_FOO ()     { 1 << 8 }
sub FLAG_BAR ()     { 1 << 9 }
sub FLAG_MASK ()    { FLAG_FOO | FLAG_BAR }

sub OPT_GLARCH ()   { (0x1B58 & FLAG_MASK) == 0 }
sub GLARCH_VAL ()   {
    if (OPT_GLARCH) { return 23 }
```

```
    else            { return 42 }
}

sub N () { int(GLARCH_VAL) / 3 }
BEGIN {            # компилятор выполнит этот блок на этапе компиляции
    my $prod = 1; # сохраняющая значение локальная переменная
    for (1 .. N) { $prod *= $_ }
    sub NFACT () { $prod }
}
```

В последнем примере вместо функции NFACT будет выполнена подстановка, потому что функция имеет пустой прототип, а возвращаемая переменная не изменяется функцией – более того, эту переменную в принципе нельзя изменить, поскольку она находится в лексической области видимости. Поэтому компилятор заменит вызовы NFACT значением, вычисленным на этапе компиляции, раз декларация заключена в блок BEGIN.

Если переопределить функцию, которая имела возможность подстановки, Perl обязательно предупредит об этом. (Благодаря данному предупреждению можно узнать, выполняет ли компилятор подстановку той или иной подпрограммы.) Данное предупреждение считается достаточно важным, чтобы его нельзя было отключить, так как ранее скомпилированные вызовы функции по-прежнему будут использовать старое значение. Если позднее потребуется переопределить функцию, следует исключить возможность сделать ее подставляемой путем удаления прототипа () (что изменит семантику вызова, поэтому будьте осторожны) или помешать механизму подстановки иным способом, например:

```
sub not_inlined () {
    return 23 if $$;
}
```

Более подробно о том, что происходит в фазах компиляции и выполнения программы, читайте в главе 16.

Предосторожности при использовании прототипов

Вероятно, прототипы лучше применять к новым функциям, а не прикручивать к уже существующим задним числом. Это шаблоны контекста, а не прототипы ANSI C, поэтому будьте внимательны, чтобы незаметно не изменить контекст. Предположим, что функция должна принимать только один параметр, как в следующем примере:

```
sub func ($) {
    my $n = shift;
    print "вы передали мне $n\n"
}
```

Это превращает функцию в унарный оператор (подобно встроенной функции rand) и изменяет порядок интерпретации аргументов функции компилятором. С новым прототипом функция принимает единственный аргумент в скалярном контексте вместо многих аргументов в списочном контексте. Если она где-то вызывалась с массивом или списком, даже если этот массив или список содержали единственный элемент, то там, где она раньше нормально работала, теперь мы будем получать совершенно иные результаты:

```
func @foo,              # количество элементов @foo
func split /:/;         # количество возвращаемых полей
func "a", "b", "c";     # передается только "a", "b" и "c" отбрасываются
func("a", "b", "c");    # внезапно: ошибка компилятора!
```

В начале списка аргументов неявно передается скаляр, что может оказаться, скажем мягко, несколько неожиданным. Прежний массив @foo**, содержавший один элемент, не передается. Вместо него** func **получит** 1 **(число элементов в** @foo**). А** split **при вызове в скалярном контексте испортит весь список параметров** @_**. В третьем примере, поскольку** func **назначен прототип унарного оператора, передано будет только** "a"**; а значение, возвращаемое** func**, будет отброшено, потому что оператор «запятая» продолжит вычисление следующих двух элементов и вернет** "c"**. В последнем примере во время компиляции пользователь столкнется с синтаксической ошибкой там, где раньше ее не было.**

Если в новой программе потребуется создать оператор, принимающий только скалярную переменную, а не прежнее скалярное выражение, в прототипе можно указать, что функция принимает *ссылку* на скаляр:

```
sub func (\$) {
    my $nref = shift;
    print "вы передали мне $$nref\n";
}
```

Теперь компилятор разрешит передавать лишь объекты, имена которых предваряются символом $:

```
func @foo;              # ошибка компиляции, есть @, нужен $
func split/:/           # ошибка компиляции, есть функция, нужен $
func $s;                # здесь порядок -- получен символ $
func $a[3];             # и здесь
func $h{stuff}[-1];     # и даже здесь
func 2+5;               # скалярное выражение, ошибка компиляции
func ${ \(2+5) };       # порядок, но это лекарство хуже болезни
```

При неосторожном обращении прототипы могут стать источником неприятностей. Но при внимательном отношении с их помощью можно творить чудеса. Конечно, это сильное средство и пользоваться им надо с умом, чтобы сделать мир лучше.

Прототипы встроенных функций

Для справки в табл. 7.2 перечислены текущие прототипы встроенных функций v5.14, которые могут быть переопределены.

Таблица 7.2. Прототипы встроенных функций

Прототип	Ключевые слова
()	and, break, continue, dump, endgrent, endhostent, endnetent, endprotoent, endpwent, endservent, fork, getgrent, gethostent, getlogin, getnetent, getppid, getprotoent, getpwent, getservent, or, setgrent, setpwent, time, times, wait, wantarray
(_)	abs, alarm, chr, chroot, cos, exp, fc, hex, int, lc, lcfirst, length, log, oct, ord, quotemeta, readlink, readpipe, ref, rmdir, sin, sqrt, uc, ucfirst
(;$)	caller, chdir, exit, getpgrp, gmtime, localtime, rand, reset, sleep, srand, umask

Таблица 7.2 (продолжение)

Прототип	Ключевые слова
(;*)	close, eof, getc, readline, select, tell, write
(;+)	pop, shift
(@)	chmod, chown, die, kill, reverse, unlink, utime, warn
(_,$)	mkdir
(;$$)	setpgrp
($)	getgrgid, getgrnam, gethostbyname, getnetbyname, getprotobyname, getprotobynumber, getpwnam, getpwuid, setnostent, setnetent, setprotoent, setservent
(*)	closedir, fileno, getpeername, getsockname, lstat, readdir, rewinddir, stat, telldir
(+)	each, keys, values
(\$)	lock
(\%)	dbmclose
(\[$@%*])	tied, untie
($;$)	bless, unpack
(*;$)	binmode
(*;$@)	open
(+;$$@)	splice
($$)	atan2, crypt, gethostbyaddr, getnetbyaddr, getpriority, getservbyname, getservbyport, link, msgget, rename, semop, symlink, truncate, waitpid
($@)	formline, join, pack, sprintf, syscall
(+@)	push, unshift
(*$)	bind, connect, flock, listen, opendir seekdir, shutdown
(**)	accept, pipe
($$;$)	index, rindex
($$;$$)	substr
(*$;$$)	syswrite
(\[$@%*]$@)	tie
($$$)	msgctl, msgsnd, semget, setpriority, shmctl, shmget, vec
(*$$)	fcntl, getsockopt, ioctl, seek, sysseek
(\%$$)	dbmopen
(*$$;$)	send, sysopen
(*\$$;$)	read, sysread
($$$$)	semctl, shmread, shmwrite
(*$$$)	setsockopt, socket
(*\$$$)	recv
($$$$$)	msgrcv
(**$$$)	socketpair

Атрибуты подпрограмм

С объявлением или определением подпрограммы можно связать список атрибутов. Разделителями в таком списке служат пробельные символы или двоеточия, и обрабатывается он так, как если бы была встречена инструкция `use attributes`. **Подробнее о применении прагмы** `use attributes` **рассказывается в главе 29. Есть два стандартных атрибута подпрограмм:** `method` **и** `lvalue`.

Атрибут method

Атрибут `method` **может использоваться самостоятельно:**

```
sub afunc . method {       }
```

В настоящее время его действие сводится к подавлению вывода предупреждения `"Ambiguous call resolved as CORE::%s"` **(неоднозначный вызов разрешен как** `CORE::%s`**). (Возможно, когда-нибудь мы придадим ему дополнительный смысл.)**

Пользователь может расширять систему атрибутов, что дает возможность создавать собственные имена атрибутов. Новые атрибуты должны удовлетворять правилам образования имен простых идентификаторов (никаких знаков пунктуации, кроме символа «_»). К ним может прилагаться список параметров, для которого в настоящее время выполняется толькс проверка сбалансированности вложенных скобок.

Примеры допустимого синтаксиса (даже если атрибуты неизвестны):

```
sub fnord (&\%)   switch(10,foo(7,3)) . expensive
sub plugh () : Ugly('\(") :Bad;
sub xyzzy : _5x5 { ... }
```

Примеры неправильного синтаксиса:

```
sub fnord   switch(10,foo(); # непарные скобки
sub snoid : Ugly('(');       # непарные скобки
sub xyzzy : 5x5;             # "5x5" не является идентификатором
sub plugh : Y2::north;       # "Y2::north" не является простым идентификатором
sub snurt : foo + bar;       # "+" не является двоеточием или пробелом
```

Список атрибутов передается в код, который связывает их с подпрограммой, как список строковых констант. Точное представление о том, как он работает (или не работает), можно получить экспериментальным путем. Текущее состояние дел со списками атрибутов и обработкой таких списков можно получить из *attributes*(3).

Атрибут lvalue

Подпрограмма может возвращать модифицируемое скалярное значение, но только если объявить, что она возвращает l-значение:

```
my $val;
sub canmod . lvalue {
    $val;
}
```

```
sub nomod {
    $val;
}

canmod() = 5; # присваивание переменной $val
nomod()  = 5; # ОШИБКА
```

При передаче параметров подпрограмме, возвращающей l-значение, обычно нужны скобки для устранения неоднозначности типа присваиваемого значения:

```
canmod $x  = 5; # сначала присвоит переменной значение 5 переменной $x;
canmod 42  = 5; # нельзя изменить константу - ошибка компиляции
canmod($x) = 5; # здесь все в порядке
canmod(42) = 5; # и здесь тоже
```

Эту особенность можно обойти, если подпрограмма принимает всего один аргумент. Когда функция объявлена с прототипом ($), на этапе синтаксического анализа она получит приоритет именованного унарного оператора. Поскольку именованные унарные операторы имеют более высокий приоритет, чем присваивание, надобность в скобках отпадает. (Целесообразность этого оставим определять полиции стилевых нравов.)

Если подпрограмма не имеет аргументов (т.е. ей назначен прототип ()), можно, не вызывая неоднозначности, сказать:

```
canmod = 5;
```

Это допустимо, потому что терм не может начинаться с символа =. Аналогично можно опускать скобки в вызовах методов, возвращающих l-значение, если им не передаются аргументы:

```
$obj->canmod = 5;
```

Обещаем, что эти две конструкции сохранятся в будущих версиях Perl. Они удобны, когда нужно обернуть (wrap) атрибуты объекта вызовами методов (чтобы их можно было наследовать, как методы, но обращаться, как к переменным).

Скалярный или списочный контекст подпрограммы, возвращающей l-значение, и правой части выражения присваивания определяется так, как если бы вызов подпрограммы был заменен скаляром. Например:

```
data(2,3) = get_data(3,4);
```

Здесь обе подпрограммы вызываются в скалярном контексте, в то время как здесь:

```
(data(2,3)) = get_data(3,4);
```

и здесь:

```
(data(2),data(3)) = get_data(3,4);
```

все подпрограммы вызываются в списочном контексте.

В текущей реализации из подпрограмм с атрибутом `lvalue` нельзя возвращать массивы и хеши непосредственно, зато всегда можно вернуть ссылку.

8

Ссылки

По причинам как практическим, так и философским, Perl всегда тяготел к плоским, линейным структурам данных. А для многих задач именно это и требуется.

Допустим, требуется построить простую таблицу (двумерный массив) с антропометрическими данными – возраст, цвет глаз и вес – для группы людей. Для этого можно сначала создать массивы, соответствующие отдельным людям:

```
@john = (47, "темно-карие", 186);
@mary = (23, "карие", 128);
@bill = (35, "голубые", 157);
```

Затем – один общий массив с именами остальных массивов:

```
@vitals = ('john', 'mary', 'bill');
```

Чтобы изменить цвет глаз Джона на «красные» после ночи, проведенной в увеселительных заведениях, надо иметь возможность изменить содержимое массива @john, используя только простую строку "john". Это основная проблема *косвенной адресации* (*indirection*), которая в разных языках решается по-разному. В языке C преобладающей формой косвенной адресации является указатель, который позволяет хранить адрес одной переменной в другой. В Perl преобладающей формой косвенной адресации является *ссылка* (*reference*).

Что такое ссылка?

В нашем примере $vitals[0] имеет значение "john". То есть данный элемент содержит строку имени другой (глобальной) переменной. Мы говорим, что первая переменная *ссылается* на вторую, и эта ссылка называется *символической* (*symbolic*), поскольку Perl приходится искать @john в таблице символов. (Символические ссылки на переменные аналогичны символическим ссылкам в файловой системе.) О символических ссылках мы будем говорить далее в этой главе.

Ссылки второго типа называются *жесткими* (*hard*), именно их чаще всего применяют программисты на Perl для решения задач косвенной адресации (иногда с не

очень приятными последствиями). Жесткими мы называем их потому, что они надежные и прочные, а не потому, что с ними сложно обращаться. Если хотите, жесткие ссылки можно считать настоящими ссылками, а символические – поддельными. Разница такая же, как между настоящей дружбой и похвальбой знакомством с известным человеком. Если мы не уточняем, какую ссылку имеем в виду, речь идет о жесткой ссылке. На рис. 8.1 изображена переменная $bar, ссылающаяся на скаляр с именем $foo и значением "bot".

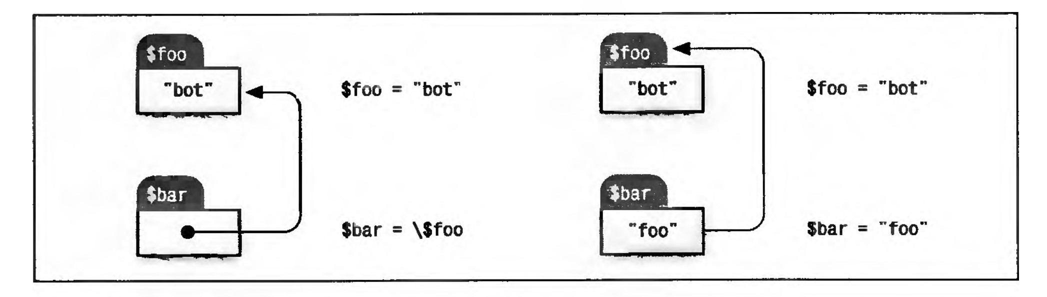


Рис. 8.1. Жесткая ссылка и символическая ссылка

В отличие от символических ссылок, настоящая ссылка ссылается не на имя переменной (которое просто хранит значение), а на фактическое значение, некоторый внутренний фрагмент данных. Подходящего слова для него нет, поэтому мы называем его *объектом ссылки* (*referent*). Предположим, что создана жесткая ссылка на массив @array, имеющий лексическую область видимости. Как эта жесткая ссылка, так и соответствующий ей объект ссылки будут существовать, даже когда @array выйдет из области видимости. Объект ссылки уничтожается, только когда ликвидированы все ссылки на него.

Объект ссылки не имеет собственного имени как такового, существуют только ссылки на этот объект. Если посмотреть на это иначе, то имя каждой переменной Perl «проживает» в той или иной таблице имен и соответствует ровно одной жесткой ссылке на связанный с этим именем (и в других отношениях безымянный) объект ссылки. Объект ссылки может быть простым значением, как число или строка, или сложным – как массив или хеш. В любом случае переменную с ее значением связывает ровно одна ссылка. На тот же объект ссылки можно создать другие ссылки, но переменная не будет об этом знать (или беспокоиться).[1]

Символическая ссылка – это простая строка, служащая именем некоторого элемента таблицы имен в пакете. Это не столько какой-то особый тип, сколько способ обращения со строкой. Но жесткая ссылка – это зверь совсем иной породы. Это третья разновидность фундаментальных скалярных типов данных (два других – строки и числа). Чтобы ссылаться на объект, жесткой ссылке не нужно имя, и, самое главное, совершенно нормально, если такого имени вообще нет. Такие совершенно безымянные объекты ссылок называются *анонимными*; мы расскажем о них далее, в разделе «Анонимные данные».

---

[1] Если вам это интересно – прочитать счетчик ссылок для объекта можно с помощью модуля Devel::Peek, поставляемого с Perl.

В терминологии данной главы, чтобы *сослаться* (*to reference*) на значение, нужно создать на него жесткую ссылку. (Этот акт созидания выполняет специальный оператор.) Созданная при этом ссылка является простым скаляром, который ведет себя как любой другой скаляр. *Разыменовать* (*dereference*) этот скаляр означает использовать ссылку для доступа к объекту ссылки. Как создание ссылки, так и разыменование происходят только при явном обращении к определенному механизму; неявное создание и разыменование ссылок в Perl 5 никогда не происходит. Ну, почти никогда.[1]

При вызове функции *может* использоваться семантика неявной передачи по ссылке, если эта семантика определена в прототипе. В таком случае ссылка передается функции неявным образом, но разыменовывать ее внутри функции приходится явно. См. раздел «Прототипы» в главе 7. И, если уж говорить совсем честно, неявное разыменование ссылок все-таки случается, когда используются некоторые виды дескрипторов файлов, но это делается для поддержки обратной совместимости и происходит прозрачно для обычного пользователя. Две встроенные функции, `bless` и `lock`, принимают в качестве аргумента ссылку и неявно разыменовывают ее, чтобы творить волшебство над объектом этой ссылки. И, наконец, начиная с Perl версии v5.14, встроенные функции, оперирующие массивами и хешами[2], теперь принимают ссылки на объекты соответствующих типов и автоматически разыменовывают их при необходимости. Но если оставить в стороне эти признания, то базовый принцип все же состоит в том, что Perl воздерживается от участия в иерархии косвенной адресации, созданной программистом.

Ссылка может указывать на любую структуру данных. Поскольку ссылки являются скалярами, их можно сохранять в массивах и хешах, создавая тем самым массивы массивов, массивы хешей, хеши массивов, массивы хешей и функций и т.д. Примеры тому есть в главе 9.

Но имейте в виду, что внутреннее представление массивов и хешей в Perl является одномерным. То есть, их элементы могут хранить только скалярные значения (строки, числа и ссылки). Под термином «массив массивов» мы подразумеваем «массив ссылок на массивы», так же как, говоря «хеш функций», мы в действительности имеем в виду «хеш ссылок на подпрограммы». А раз в Perl ссылки являются единственным способом реализации подобных структур, то более короткий, но менее точный термин не настолько неточен, чтобы быть неверным, потому не следует его отвергать, если только вы не придаете особого значения такого рода вещам.

Создание ссылок

Есть несколько способов создания ссылок, и мы опишем большинство этих способов, прежде чем объясним, как использовать (разыменовывать) полученные ссылки.

---

1 Чтобы запутать вас еще больше, скажем, что в Perl 6 это происходит практически всегда.

2 `keys`, `values`, `each`, `pop`, `push`, `shift`, `unshift` и `splice`.

Оператор обратной косой черты

Ссылку на любую именованную переменную или подпрограмму можно создать с помощью обратной косой черты. (Ее можно также использовать с анонимными скалярными значениями, например 7 или "camel", но это редко когда пригождается.) Этот оператор действует подобно оператору & (взятие адреса) в C, по крайней мере, на первый взгляд.

Вот несколько примеров:

```
$scalarref = \$foo;
$constref  = \186_282.42;
$arrayref  = \@ARGV;
$hashref   = \%ENV;
$coderef   = \&handler;
$globref   = \*STDOUT;
```

Оператор обратной косой черты может сделать больше, чем просто создать одну ссылку. Если применить его к списку, он создаст целый список ссылок. Подробности читайте в разделе «Другие приемы использования жестких ссылок».

Анонимные данные

В примерах выше оператор обратной косой черты просто создает дубликаты ссылок, уже связанных с именами переменных, за одним исключением: 186_282.42 является простым значением, а не именованной переменной. Это пример одного из вышеупомянутых *анонимных* объектов ссылки. Доступ к анонимным объектам ссылки осуществляется только с помощью ссылок. Данный объект представляет собой число, но можно также создать анонимный массив, хеш и подпрограмму.

Формирование анонимных массивов

Ссылку на анонимный массив можно создать с помощью квадратных скобок:

```
$arrayref = [1, 2, ['a', 'b', 'c', 'd']];
```

Здесь сформирован анонимный массив из трех элементов, последний из которых является ссылкой на анонимный массив с четырьмя элементами (как изображено на рис. 8.2). (Для доступа к этим элементам можно использовать синтаксис многомерных массивов, описываемый ниже. Например, выражение $arrayref->[2][1] вернет значение 'b'.)

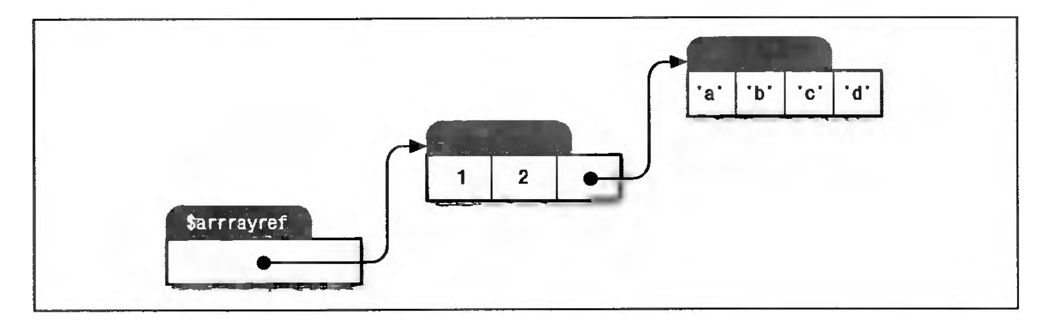


Рис. 8.2. *Ссылка на массив, третий элемент которого является ссылкой на массив*

Теперь у нас есть способ представить таблицу, приведенную в начале главы:

```
$table = [ [ "john", 47, "темно-карие", 186],
           [ "mary", 23, "карие",       128],
           [ "bill", 35, "голубые",     157] ];
```

Квадратные скобки действуют подобным образом только там, где анализатор Perl предполагает наличие терма в выражении. Не следует путать их с квадратными скобками в таких выражениях, как `$array[6]`, хотя мнемоническая ассоциация с массивами является намеренной. Внутри строки, заключенной в двойные кавычки, квадратные скобки не образуют анонимных массивов, а становятся литеральными символами строки. (Квадратные скобки все же действуют, когда обозначают индексы массивов в строках, иначе нельзя было бы выводить строковые значения, такие как `"VAL=$array[6]\n"`. И если уж совсем откровенно, формирование анонимных массивов все же можно протащить в строки, но только если они встроены в более сложные выражения, которые подвергаются интерполяции. К этой замечательной возможности мы еще вернемся в этой главе, поскольку для ее применения требуется как создание ссылок, так и разыменование оных.)

Формирование анонимных хешей

Ссылку на анонимный хеш можно создать с помощью фигурных скобок:

```
$hashref = {
    "Адам"    => "Ева",
    "Клайд"   => $bonnie,
    "Антоний" => "Клео" . "патра",
}
```

В значениях хеша (но не в ключах) можно смело смешивать формировать другие анонимные массивы, хеши и подпрограммы, создавая сколь угодно сложные структуры.

Теперь у нас есть еще один способ представить таблицу, приведенную в начале главы:

```
$table = {
            "john" => [ 47, "темно-карие"  186 ],
            "mary" => [ 23, "карие",       128 ],
            "bill" => [ 35  "голубые"      157 ],
};
```

Это хеш массивов. Выбор наиболее уместной структуры данных – дело непростое, и этому вопросу посвящена следующая глава. В качестве рекламы покажем, как создать хеш хешей для представления нашей таблицы:

```
$table = {
            "john" => { age    => 47,
                        eyes   => "темно-карие",
                        weight => 186
                      },
            "mary" => { age    => 23,
                        eyes   => "карие",
                        weight => 128
                      },
            "bill" => { age    => 35,
```

```
                              eyes   => "голубые",
                              weight => 157,
                            },
};
```

Фигурные скобки, как и квадратные, имеют специальное значение только там, где анализатор Perl предполагает наличие терма в выражении. Не следует путать их с фигурными скобками в выражениях вида $hash{key} – хотя мнемоническая ассоциация с массивами и в этом случае является намеренной. При использовании фигурных скобок в строках действуют те же правила.

Есть еще одно правило, не распространяющееся на квадратные скобки. Поскольку фигурные скобки используются также для других целей (в том числе для разграничения блоков), иногда приходится устранять неоднозначность фигурных скобок в начале оператора, помещая перед ними + или return, чтобы Perl понял, что открывающая скобка не начинает блок. Например, чтобы функция создала новый хеш и вернула ссылку на него, можно использовать такие варианты:

```
sub hashem {        { @_ } } # НЕВЕРНО, просто вернет @_.
sub hashem {       +{ @_ } } # Ok.
sub hashem { return { @_ } } # Ok.
```

Формирование анонимных подпрограмм

Создать ссылку на анонимную подпрограмму позволяет объявление sub без имени подпрограммы:

```
$coderef = sub { print "Boink!\n" }; # вызов &$coderef выведет "Boink!"
```

Обратите внимание на точку с запятой, которая требуется для завершения выражения. (Точка с запятой не обязательна после более распространенной формы объявления и определения именованной подпрограммы, sub *NAME* {}.) Объявление sub {} без имени является не столько объявлением, сколько оператором – как {} или eval {} – за исключением того, что код внутри блока не выполняется немедленно. Вместо этого генерируется ссылка на код – в нашем примере эта ссылка сохраняется в $coderef. Но сколько бы раз ни выполнялась приведенная выше строка, $coderef будет ссылаться на одну и ту же безымянную процедуру.[1]

Конструкторы объектов

Подпрограммы также могут возвращать ссылки. Может звучать банально, но иногда мы *обязаны* использовать подпрограммы для создания ссылок, а не создавать их напрямую. В частности, особые подпрограммы, называемые *конструкторами*, создают или возвращают ссылки на объекты классов. Объект класса – это просто ссылка особого типа, связанная с определенным классом, и конструкторы умеют создавать эту связь. Они берут обычный объект ссылки и превращают его в объект класса с помощью оператора bless, поэтому можно считать, что объект класса – это «освященная ссылка» (blessed reference). Ничего религиозного в этом нет. Поскольку класс – это тип, определяемый пользователем, обработка объекта

---

[1] Анонимная подпрограмма только одна, однако она может использовать несколько наборов лексических переменных – в зависимости от того, когда генерируется ссылка на подпрограмму. О таких наборах мы расскажем далее, в разделе «Замыкания».

ссылки с помощью bless просто делает его экземпляром типа, определенного пользователем. Конструкторам часто дают имя new – особенно программисты на языках C++ и Java – но в Perl их можно называть как угодно.

Конструкторы можно вызывать любым из следующих способов:

```
$objref = Doggie::->new(Tail => 'короткий', Ears => 'длинные');   #1
$objref = new Doggie:: Tail => 'короткий', Ears => 'длинные';     #2
$objref = Doggie->new(Tail => 'короткий', Ears => 'длинные');     #3
$objref = new Doggie Tail => 'короткий', Ears => 'длинные';       #4
```

Первый и второй способы одинаковы. Оба вызывают функцию new из модуля Doggie. Третий и четвертый способы действуют так же, как первые два, но они несколько двусмысленны: анализатор запутается, если определить собственную подпрограмму с именем Doggie. (Поэтому в именах подпрограмм обычно придерживаются символов нижнего регистра, а символы верхнего регистра используют в именах модулей.) Четвертый способ также приведет к путанице, если определить собственную подпрограмму new и не выполнить операторы require или use для модуля Doggie, каждый из которых имеет эффект объявления модуля. Всегда объявляйте свои модули, если хотите пользоваться вариантом 4. (И остерегайтесь «бродячих» подпрограмм Doggie.)

Объектам Perl посвящена глава 12.

Ссылки на дескрипторы

Ссылки на дескрипторы файлов или дескрипторы каталогов можно создавать, ссылаясь на typeglob с такими же именами:

```
splutter(\*STDOUT);

sub splutter {
    my $fh = shift;
    say $fh "her um well a hmmm";
}

$rec = get_rec(\*STDIN);
sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

Для передачи дескрипторов файлов можно применять голые (простые) typeglob: в примере выше вместо *STDOUT и *STDIN можно было бы использовать *STDOUT или *STDIN.

Хотя typeglob и ссылки на typeglob обычно можно использовать взаимозаменяемо, в некоторых случаях это недопустимо. Простые typeglob нельзя катапультировать в царство объектное посредством bless, а ссылки на typeglob нельзя вернуть из области видимости локализованного typeglob.

Раньше использовался следующий прием создания новых дескрипторов файлов при открытии файлов по списку:

```
for $file (@names) {
    local *FH;
```

```
        open(*FH, $file) || next;
        $handle{$file} = *FH;
    }
```

Такой подход по-прежнему в силе, но часто разумнее позволить неопределенной переменной автоматически «оживить» анонимный typeglob, т.е. выполнить ***автовивификацию***[1]:

```
    for $file (@names) {
        my $fh;
        open($fh, $file) || next;
        $handle{$file} = $fh;
    }
```

Всякий раз, когда вместо простого дескриптора файла используется переменная, содержащая дескриптор, создается косвенный дескриптор. Не имеет значения, что вы используете: строки, typeglob, ссылки на typeglob или более экзотические объекты ввода/вывода. Вы просто используете скаляр, который так или иначе будет интерпретирован в качестве дескриптора файла. В большинстве случаев почти не имеет значения, что выбрать – typeglob или ссылку на typeglob. Как уже отмечалось выше, при этом включаются определенные волшебные механизмы неявного разыменования.

Ссылки на таблицы символов

В редких случаях тип требуемой ссылки может быть неизвестен во время написания программы. Такую ссылку можно создать с помощью особого синтаксиса, который ласково называют синтаксисом `*foo{THING}`. Выражение вида `*foo{THING}` возвращает ссылку на элемент *THING* в `*foo`,[2] который является записью в таблице имен, содержащей значения `$foo`, `@foo`, `%foo` сотоварищи.

```
    $scalarref = *foo{SCALAR};   # То же, что и \$foo
    $arrayref  = *ARGV{ARRAY};   # То же, что и \@ARGV
    $hashref   = *ENV{HASH};     # То же, что и \%ENV
    $coderef   = *handler{CODE}; # То же, что и \&handler
    $globref   = *foo{GLOB};     # То же, что и \*foo
    $ioref     = *STDIN{IO};     # Э-э
    $formatref = *foo{FORMAT};   # И снова э-э..
```

Здесь все ясно без объяснений, за исключением последних двух строк. `*foo{FORMAT}` возвращает объект, объявленный с помощью оператора `format`. Такая ссылка не представляет особого интереса.

А вот `*STDIN{IO}` возвращает фактический внутренний объект `IO::Handle`, хранящийся в typeglob, т.е. ту часть typeglob, которая представляет немалый интерес для различных функций ввода/вывода. Для совместимости с ранними версиями Perl обозначение `*foo{FILEHANDLE}` когда-то служило синонимом более привычного `*foo{IO}`, но в настоящее время использовать его не рекомендуется.

---

[1] Автовивификация – отличительная особенность языка Perl, связанная с динамическим конструированием структур данных. – *Прим. науч. ред.*

[2] Развернутое толкование и историю термина «foo» можно найти в словаре сленга Jargon File (*http://www.jargondb.org/*). – *Прим. ред.*

Теоретически *HANDLE{IO} можно использовать везде, где допустимы *HANDLE или *HANDLE, например для передачи дескрипторов в подпрограмму или из нее или для сохранения в более развитых структурах данных. (На практике все же есть некоторые шероховатости, которые предстоит сгладить.) Преимущество таких ссылок заключается в том, что они ограничивают доступ лишь интересующими вас объектами ввода/вывода, исключая риск затереть другие данные typeglob, присвоив значение по typeglob-ссылке (хотя если всегда выполнять присваивание скалярной переменной, а не typeglob, все будет в порядке). Есть и недостатки – скажем, на сегодняшний день нет механизма автовивификации для таких ссылок.[1]

```
splutter(*STDOUT); splutter(*STDOUT{IO});

sub splutter {
    my $fh = shift; print $fh "her um well a hmmm\n";
}
```

Оба вызова splutter() выведут "her um well a hmmm".

Конструкция *foo{*THING*} возвращает undef, если компилятор еще не встречал указанный аргумент *THING*, кроме случаев, когда *THING* представляет собой SCALAR. Дело в том, что *foo{SCALAR} возвращает ссылку на анонимный скаляр, даже если компилятор еще не встречал $foo. (Perl всегда добавляет скаляр к любому typeglob для оптимизации кода по размеру. Но не следует полагать, что так будет и в последующих версиях.)

Неявное создание ссылок

Выше вы видели некоторые замысловатые примеры *автовивификации* – и это последний метод создания ссылок, который, в сущности, методом не является. Ссылки нужного типа просто возникают при разыменовании их в контексте l-значения, предполагающего их существование. Это очень удобно и это как раз ожидаемое поведение. Мы еще вернемся к этой теме далее в этой главе, когда речь пойдет о разыменовании созданных ссылок. Кстати, похоже, что мы как раз добрались до этого места.

Использование жестких ссылок

Способов использования, или *разыменования* (*dereference*), ссылок ничуть не меньше, чем способов их создания. Есть только один главный принцип: Perl не создает и не разыменовывает ссылки неявно.[2] Если скаляр содержит ссылку, он ведет себя как простой скаляр. Он не начинает чудесным образом вести себя как массив, или хеш, или подпрограмма: об этом ему нужно сказать явно, путем разыменования.

---

[1] В настоящее время open my $fh автовивифицирует typeglob, а не объект IO::Handle, но когда-нибудь мы, возможно, исправим это, поэтому не следует ориентироваться на существующее положение дел и считать, что автовивифицированный функцией open объект – обязательно typeglob.

[2] Мы уже признавались, что это маленькая ложь, и повторяться не будем.

Использование переменной в качестве имени переменной

Встретив скаляр, скажем $foo, следует размышлять о нем, как о «скалярном значении имени foo». То есть где-то в таблице имен содержится запись foo, а *разыменовывающий символ* $ (sigil) дает возможность увидеть скалярное значение в ней. Если это значение является ссылкой, можно посмотреть, что находится *по этой ссылке*, предварив обращение к скаляру еще одним разыменовывающим символом (т.е. разыменовав $foo). Или, если посмотреть на это с другой стороны, можно заменить литерал foo в скаляре $foo скалярной переменной, указывающей на фактический объект ссылки. Это верно для переменной любого типа, т.е. $$foo является скалярным значением, на которое ссылается $foo, @$bar – массивом, на который ссылается $bar, %$glarch – хешем, на который ссылается $glarch, и т.д. Таким образом, перед любой простой скалярной переменной можно поместить дополнительный разыменовывающий символ и разыменовать ее:

```
$foo         = "трехгорбый";
$scalarref   = \$foo;        # $scalarref теперь ссылка на $foo
$camel_model = $$scalarref; # $camel_model теперь"трехгорбый"
```

Еще несколько примеров разыменования:

```
$bar = $$scalarref;

push(@$arrayref, $filename);
$$arrayref[0] = "January";             # Изменит первый элемент @$arrayref
@$arrayref[4..6] = qw/May June July/;# Изменит несколько элементов @$arrayref

%$hashref = (KEY => "ЗВЕНИТ", BIRD => "ПОЕТ");  # Инициализирует весь хеш
$$hashref{KEY} = "ЗНАЧЕНИЕ";                     # Изменит одну пару ключ/значение
@$hashref{"КЛЮЧ1","КЛЮЧ2"} = ("ЗНАЧ1","ЗНАЧ2"); # Изменит еще две пары

&$coderef(1,2,3);

say $handleref "вывод";
```

Эта форма разыменования может применяться только к простым скалярным переменным (без индексов). Это значит, что разыменование выполняется *до* (имеет более высокий приоритет) поиска в массиве или хеше. Попробуем с помощью фигурных скобок пояснить, что мы имеем в виду: выражение $$arrayref[0] эквивалентно ${$arrayref}[0] и возвращает первый элемент массива, на который ссылается $arrayref. Это совсем не то же самое, что выражение ${$arrayref[0]}, разыменовывающее первый элемент (вероятно, несуществующего) массива с именем @arrayref. Аналогично, выражение $$hashref{KEY} эквивалентно ${$hashref}{KEY}, и не имеет отношения к выражению ${$hashref{KEY}}, разыменовывающему элемент (вероятно, несуществующего) хеша с именем %hashref. Не видать вам счастья, пока вы этого не поймете.

Комбинируя разыменовывающие символы, можно работать на различных уровнях иерархии ссылок и разыменований. Следующий пример выведет "howdy":

```
$refrefref = \\\"howdy";
print $$$$refrefref;
```

Можно считать, что знаки доллара действуют справа налево. Но в начале цепочки все же должна быть простая скалярная переменная без индекса. Есть, правда, более замысловатый способ, которым мы уже тайком пользовались выше и о котором расскажем в следующем разделе.

Использование блока в качестве имени переменной

Разыменовывать можно не только имя простой переменной, но и содержимое блока. Всюду, где буквенно-цифровой идентификатор играет роль имени переменной или подпрограммы, его можно заменить блоком, возвращающим ссылку подходящего типа. Иными словами, все приведенные выше примеры можно избавить от двусмысленности следующим образом:

```
$bar = ${$scalarref};
push(@{$arrayref}, $filename);
${$arrayref}[0] = "January";
@{$arrayref}[4..6] = qw/May June July/;
${$hashref}{"KEY"} = "ЗНАЧЕНИЕ";
@{$hashref}{"КЛЮЧ1","КЛЮЧ2"} = ("ЗНАЧ1","ЗНАЧ2");
&{$coderef}(1,2,3),
```

не говоря уже о:

```
$refrefref = \\\"howdy";
print ${${${$refrefref}}};
```

В таких простых примерах глупо использовать фигурные скобки, но блок может содержать произвольные выражения и, в частности, выражения с индексами. В следующем примере предполагается, что `$dispatch{$index}` содержит ссылку на подпрограмму (такие ссылки иногда называют «coderef» – ссылкой на код). Подпрограмма вызывается с тремя аргументами:

```
&{ $dispatch{$index} }(1, 2, 3);
```

Здесь блок необходим. Без внешней пары фигурных ссылкой на код будет считаться `$dispatch`, а не `$dispatch{$index}`.

Использование оператора «стрелка»

Третий способ разыменования, с использованием инфиксного оператора `->`, предназначен для ссылок на массивы, хеши или подпрограммы. Эта форма служит упрощению синтаксиса доступа к элементам массивов и хешей, а также косвенного вызова подпрограмм.

Тип разыменования определяется правым операндом, т.е. тем, что следует непосредственно за стрелкой. Если за стрелкой следует квадратная или фигурная скобка, левый операнд считается ссылкой на массив или хеш, соответственно, а выражение справа – индексом. Если за стрелкой следует круглая скобка, то левый операнд считается ссылкой на подпрограмму, которая должна вызываться с параметрами в круглых скобках справа.

Операторы в каждой из следующих триад эквивалентны между собой и соответствуют трем типам обозначений, с которыми мы познакомились. (Эквивалентные элементы выровнены при помощи пробелов.)

```
$  $arrayref  [2] = "Dorian";   #1
${ $arrayref }[2] = "Dorian";   #2
   $arrayref->[2] = "Dorian";   #3

$  $hashref  {KEY} = "F#major"; #1
${ $hashref }{KEY} = "F#major"; #2
   $hashref->{KEY} = "F#major"; #3

&  $coderef  (Presto => 192);   #1
&{ $coderef }(Presto => 192);   #2
   $coderef->(Presto => 192);   #3
```

Можно заметить, что начальный разыменовывающий символ отсутствует в третьей форме записи каждой тройки. Perl угадывает начальный символ, и потому его нельзя использовать для разыменования целых массивов, целых хешей или их срезов. При работе же со скалярными значениями слева от оператора -> можно расположить любое выражение, в том числе другое разыменование, поскольку операторы стрелки связываются слева направо:

```
print $array[3]->{"English"}->[0];
```

Из этого выражения можно заключить, что четвертый элемент @array **должен быть ссылкой на хеш, а значением элемента** "English" **в этом хеше должна быть ссылка на массив.**

Обратите внимание, что $array[3] **и** $array->[3] **– это разные вещи. В первом случае речь идет о четвертом элементе массива** @array**, а во втором – о четвертом элементе массива (возможно, анонимного), ссылка на который содержится в** $array**.**

Предположим теперь, что элемент $array[3] **не определен. Следующая инструкция все же допустима:**

```
$array[3]->{"English"}->[0] = "January";
```

Это один из тех упоминавшихся выше случаев, когда ссылка создается (автовивифицируется) при использовании в качестве l-значения (т.е. в момент, когда ей присваивается значение). Если элемент $array[3] **прежде был неопределен, он автоматически определяется как ссылка на хеш, и в него можно поместить значение** $array[3]->{"English"}**. После этого элемент** $array[3]->{"English"} **автоматически определяется как ссылка на массив и можно что-нибудь присвоить первому элементу этого массива. Обратите внимание, что для r-значений дело обстоит несколько иначе:** print $array[3]->{"English"}->[0] **определит только** $array[3] **и** $array[3]->{"English"}**, но не** $array[3]->{"English"}->[0]**, потому что последний элемент не является l-значением. (Автовивификацию первых двух элементов в контексте r-значения вообще можно считать ошибкой. Вероятно, мы ее когда-нибудь исправим.)**

Стрелку между квадратными или круглыми скобками либо между закрывающей квадратной или фигурной скобкой и круглой скобкой в косвенном вызове подпрограммы можно опускать. Предыдущий пример можно сократить до:

```
$dispatch{$index}(1, 2, 3);
$array[3]{"English"}[0] = "January";
```

Для обычных массивов это дает многомерные массивы, такие же, как в C:

```
$answer[$x][$y][$z] += 42;
```

Ну, хорошо, *не совсем такие*, как в C. Во-первых, C не умеет увеличивать массивы при необходимости, а Perl умеет. Кроме того, некоторые конструкции, кажущиеся похожими, интерпретируются в этих языках по-разному. В Perl следующие две инструкции делают одно и то же:

```
$listref->[2][2] = "hello";    # Вполне понятно
$$listref[2][2]  = "hello";    # Немного смущает
```

Вторая инструкция может привести в замешательство программиста на C, привыкшего использовать *a[i], подразумевая: «на что указывает *i*-й элемент a». Но в Perl пять символов ($ @ * % &) имеют приоритет более высокий, чем фигурные или квадратные скобки.[1] Поэтому за ссылку на массив принимается $$listref, а не $listref[2]. Чтобы было как в C, нужно либо написать ${$listref[2]} и заставить Perl интерпретировать $listref[2] перед разыменованием $, либо использовать стрелку ->:

```
$listref[2]->[$greeting] = "hello";
```

Использование методов объектов

Если ссылка указывает на объект, то класс, определяющий этот объект, может предоставлять методы доступа к содержимому объекта, которых следует придерживаться, если вы лишь используете класс (а не являетесь его автором). Иными словами, ведите себя хорошо и не обращайтесь с объектом, как с обычной ссылкой, даже при том, что Perl позволяет это делать в случае острой необходимости. Perl не обеспечивает принудительную инкапсуляцию. Мы не сторонники тоталитаризма. Однако мы рассчитываем на элементарную вежливость.

В обмен на вежливость вы получаете полную ортогональность между объектами и структурами данных. Любая структура данных может вести себя как объект, когда это требуется. Или не вести, когда не требуется.

Псевдохеши

Псевдохеши когда-то служили средством интерпретации массивов в виде хешей с целью имитации упорядоченных хешей. Псевдохеши оказались не самым удачным экспериментом и не существуют начиная с версии v5.10, но многие продолжают использовать более старые версии Perl, поэтому мы решили коротко упомянуть о них, хоть вам и не следует их применять. Для работы с псевдохешами применялись функции phash и new из модуля fields.

Интерфейс доступа к псевдохешам, fields::phash, не поддерживается с версии v5.10, однако функция fields::new все еще доступна. Тем не менее, вместо нее следует использовать ограниченные хеши из стандартного модуля Hash::Util.

Другие приемы работы с жесткими ссылками

Как говорилось выше, оператор обратной косой черты обычно используется с одним объектом ссылки для создания одной ссылки, но возможны и другие сценарии.

---

[1] Это не связано с приоритетами операторов. Разыменовывающие символы в Perl не являются операторами в этом смысле. Просто грамматика Perl просто запрещает предварять начальным разыменовывающим символом что-либо более сложное, чем обычная переменная или блок.

Если применить этот оператор к списку объектов ссылок, он генерирует список соответствующих ссылок. Вторая строка в следующем примере делает то же, что и первая, поскольку обратная косая черта автоматически распределяется по всему списку.

```
@reflist = (\$s, \@a, \%h, \&f);  # Список из четырех ссылок
@reflist = \($s,  @a   %h,  &f);  # То же самое
```

Если список в круглых скобках содержит ровно один массив или хеш, то интерполируются все значения массива или хеша и возвращаются ссылки на каждое из них:

```
@reflist = \(@x);                 # Интерполировать массив, затем получить ссылки
@reflist = map { \$_ } @x;        # То же самое
```

То же происходит при наличии внутренних скобок:

```
@reflist = \(@x, (@y));           # Разворачиваются только одиночные составные объекты
@reflist = (\@x, map { \$_ } @y); # То же самое
```

Если то же проделать с хешем, результат будет содержать ссылки на значения (ожидаемый результат) и ссылки на *копии* ключей (довольно неожиданный результат).

Поскольку срезы массивов и хешей представляют собой простые списки, перед любым из них можно поставить обратную косую черту и получить список ссылок. В каждой из трех последующих строк делается одно и то же:

```
@envrefs = \@ENV{"HOME", "TERM"};         # Обратная косая черта перед срезом
@envrefs = \( $ENV{HOME}, $ENV{TERM} );   # Обратная косая черта перед списком
@envrefs = ( \$ENV{HOME}, \$ENV{TERM} );  # Список из двух ссылок
```

Поскольку функции могут возвращать списки, к ним так же можно применять обратную косую черту. Если нужно вызвать несколько функций, сначала интерполируйте значения, возвращаемые функциями, в один большой список, а затем поставьте перед ним обратную косую черту:

```
@reflist = \fx();
@reflist = map { \$_ } fx();              # То же самое

@reflist = \( fx(), fy(), fz() ),
@reflist = ( \fx(), \fy(), \fz() )        # То же самое
@reflist = map { \$_ } fx(), fy(), fz();  # То же самое
```

Оператор обратной косой черты всегда предоставляет своему операнду списочный контекст, поэтому все эти функции вызываются в списочном контексте. Если же сам оператор обратной косой черты находится в скалярном контексте, в итоге будет получена ссылка на последнее значение списка, который вернула функция:

```
@reflist = \localtime(); # Ссылки на каждый из 9 элементов времени
$lastref = \localtime(); # Ссылка на признак летнего времени
```

В этом отношении обратная косая черта действует подобно именованным списочным операторам, таким как print, reverse и sort, всегда предоставляющим списочный контекст в правой части независимо от того, что происходит в левой. Как и с именованными списочными операторами, явно используйте scalar, чтобы установить скалярный контекст для последующих операндов:

```
$dateref = \scalar localtime(); # \"Tue Oct 18 07:23:50 2011"
```

Определить, на что указывает ссылка, можно с помощью оператора ref. Считайте ref оператором «typeof», который возвращает истиное значение, если его аргумент является ссылкой, и ложное в противном случае. Возвращаемое значение зависит от типа объекта ссылки. Встроенными типами являются SCALAR, ARRAY, HASH, CODE, GLOB, REF, VSTRING, IO, LVALUE, FORMAT и REGEXP плюс классы version, Regexp и IO::Handle. Проверить аргументы подпрограммы при помощи оператора ref можно так:

```
sub sum {
    my $arrayref = shift;
    warn "Не ссылка на массив" if ref($arrayref) ne "ARRAY";
    return eval join("+", @$arrayref);
}

say sum([1..100]); # 5050, формула Эйлера
```

Жесткая ссылка в строковом контексте превращается в строку, содержащую тип и адрес: SCALAR(0x1fc0e). (Обратное преобразование невозможно, поскольку при преобразовании в строку теряются данные о счетчике ссылок, а также потому, что было бы опасно позволить программе обращаться к адресу памяти, указанному в произвольной строке.)

Посредством оператора bless можно устанавливать связь между объектом ссылки и пакетом, действующим как класс объекта. В таком случае ref возвращает не внутренний тип, а имя класса. При использовании ссылки на объект в строковом контексте возвращается строка, содержащая внешний и внутренний тип, а также адрес памяти: MyType=HASH(0x20d10) или IO::Handle=IO(0x186904). Объектам и подробностям работы с ними посвящена глава 12.

Поскольку способ разыменовывания всегда определяет вид возвращаемого объекта ссылки, typeglob, хотя и содержит несколько объектов ссылки различных типов, может использоваться так же, как ссылка. То есть, выражения ${*main::foo} и ${\$main::foo} обращаются к одной и той же скалярной переменной, хотя последнее обращение более эффективно.

Такой прием позволяет вставить в строку значение, возвращаемое подпрограммой:

```
say "На этот раз моя подпрограмма вернула @{[ mysub(1,2,3) ]}."
```

Как это работает? Когда компилятор обнаруживает @{...} в двойных кавычках, то интерпретирует это выражение как блок, возвращающий ссылку. Квадратные скобки в блоке создают ссылку на анонимный массив, содержащий все, что находится в них. Поэтому на этапе выполнения mysub(1,2,3) вызывается в списочном контексте, а результаты помещаются в анонимный массив, ссылка на который возвращается внутри блока. Эта ссылка на массив немедленно разыменовывается охватывающим @{...}, а содержимое массива вставляется в строку в двойных кавычках уже в обычном порядке. Такое крючкотворство удобно использовать и с произвольными выражениями, например:

```
say "Нам нужно @{ [$n + 5] } штуковин!"
```

Но будьте осторожны: квадратные скобки создают списочный контекст для своего выражения. В данном случае это безразлично, но в предыдущем вызове mysub могло иметь значение. Когда это существенно, устанавливайте контекст явно с помощью scalar:

```
say "теперь mysub возвращает @{ [scalar mysub(1,2,3)] }.";
```

Замыкания

Выше мы говорили о создании анонимных подпрограмм с помощью конструкции `sub {}`, не содержащей имени. Можно считать, что такие подпрограммы определяются на этапе выполнения, а это означает, что они обладают такими характеристиками, как момент генерации и местонахождение определения. Во время создания подпрограммы в области видимости могут находиться одни переменные, а при ее вызове – другие.

Забудем на мгновение о подпрограммах и рассмотрим ссылку на лексическую переменную:

```
{
    my $critter = "camel";
    $critterref = \$critter;
}
```

Значением `$$critterref` останется `"camel"`, несмотря на то, что `$critter` исчезает за закрывающей фигурной скобкой. Но `$critterref` с таким же успехом может ссылаться на подпрограмму, выполняющую обращение к `$critter`:

```
{
    my $critter = "camel";
    $critterref = sub { return $critter };
}
```

Это и есть *замыкание* (*closure*), понятие, пришедшее из функционального программирования на языках LISP и Scheme.[1] Оно означает, что анонимная функция, определенная в некоторой лексической области видимости в тот или иной момент времени, при вызове извне этой области должна вести себя так, будто находится в этой области видимости. (Пурист сказал бы, что она ничего не должна – она *действительно выполняется* в этой области видимости.)

Иными словами, каждый раз гарантируется получение того же экземпляра лексической переменной, даже если другие экземпляры этой лексической переменной были созданы раньше или позже для других экземпляров этого замыкания. Это дает возможность устанавливать значения, используемые в подпрограмме, в момент ее определения, а не только при вызове.

Замыкание можно также рассматривать как способ определения шаблона подпрограммы без использования `eval`. Лексические переменные выступают в качестве параметров заполнения шаблона, что удобно для конструирования небольших фрагментов кода с отложенным выполнением. В событийно-ориентированном программировании, когда фрагменты кода связываются с событиями нажатий клавиш, щелчков мышью, появлением окон и прочими, они обычно называются *функциями обратного вызова* (*callbacks*). В качестве функций обратного вызова замыкания обладают ожидаемым поведением, даже если у вас нет ни малейшего понятия о функциональном программировании. (Обратите внимание: все, что говорилось о замыканиях, относится только к лексическим переменным `my` и `state`. Глобальные переменные продолжают работать как обычно, потому что они не создаются и не уничтожаются, как это происходит с лексическими переменными.)

---

[1] В данном контексте слово «функциональное» не должно рассматриваться как антоним слова «дисфункциональное».

Другое применение замыкания находят в *генераторах функций*, т.е. в функциях, которые создают и возвращают совершенно новые функции. Вот пример генератора функций, реализованного с помощью замыканий:

```
sub make_saying {
    my $salute = shift;
    my $newfunc = sub {
        my $target = shift,
        say "$salute, $target!";
    };
    return $newfunc;              # Возвращает замыкание
}

$f = make_saying("Howdy");        # Создает замыкание
$g = make_saying("Greetings");    # Создает еще одно замыкание

# Спустя некоторое время

$f->("world");
$g->("earthlings");
```

В результате будет выведено:

```
Howdy, world!
Greetings, earthlings!
```

Обратите внимание, что $salute продолжает ссылаться на фактическое значение, переданное в make_saying, хотя my $salute вышла из области видимости ко времени вызова подпрограммы. Для этого и предназначены замыкания. Поскольку $f и $g содержат ссылки на функции, которым при вызове требуется доступ к разным версиям $salute, эти версии автоматически сохраняются. Если теперь перезаписать $f, то *ее* версия $salute автоматически исчезнет. (Perl производит уборку, только когда вы этого не видите.)

Perl не поддерживает ссылки на методы объектов (описываемые в главе 12), но похожего эффекта можно добиться с помощью замыкания. Допустим, что нужна ссылка не просто на подпрограмму, представленную методом, но на подпрограмму, вызывающую метод определенного объекта. Объект и метод можно сохранить в лексических переменных, связанных с замыканием:

```
sub get_method_ref {
    my ($self, $methodname) = @_;
    my $methref = sub {
        # @_ внизу не та же, что вверху!
        return $self->$methodname(@_);
    };
    return $methref;
}

my $dog = new Doggie::            # новая собачонка
            Name => "Lucky",      # по кличке Везунчик,
            Legs => 3,            # у нее три ноги
            Tail => "clipped";    # и куцый хвост

our $wagger = get_method_ref($dog, 'wag');
$wagger->("tail");                # Вызовет $dog->wag("tail").
```

Теперь можно заставить Везунчика (Lucky) вилять (wag) тем, что осталось от его хвоста, даже когда лексическая переменная $dog уйдет из области видимости, а Везунчика не будет видно: глобальная переменная $wagger заставит его вилять хвостом, где бы он ни находился.

Замыкания как шаблоны функций

Использование замыкания в качестве шаблона функции позволяет сгенерировать несколько функций, действующих аналогичным образом. Допустим, нам нужен набор функций, генерирующих изменение цвета шрифта в HTML:

```
print "Будь ", red("осторожен"), " с этим ", green("огнем"), "!!!";
```

Функции red и green должны быть очень похожи. Хотелось бы дать им имена, но у замыканий нет имен, так как они – простые анонимные подпрограммы. Чтобы обойти это обстоятельство, прибегнем к хитрому трюку. Ссылку на код можно связать с существующим именем, присвоив его переменной typeglob с нужным именем функции. См. раздел «Таблицы имен» главы 10. В данном случае мы свяжем ссылку с двумя разными именами, где одно составлено из символов верхнего регистра, а другое из символов нижнего регистра:

```
@colors = qw(red blue green yellow orange purple violet);
for my $name (@colors) {
    no strict "refs";          # Разрешить символические ссылки
    *$name = *{uc $name} = sub { "<FONT COLOR='$name'> @_</FONT>" };
}
```

Теперь функции можно вызывать с именами red, RED, blue, BLUE и т.д.; при этом будут вызываться соответствующие замыкания. Этот прием уменьшает продолжительность компиляции и экономит память, а также уменьшает вероятность ошибок, поскольку проверка синтаксиса происходит во время компиляции. Важно, чтобы при создании замыкания все переменные в анонимной подпрограмме были лексическими. Это объясняет появление my в примере выше.

Это один из тех редких случаев, когда имеет смысл указать прототип для замыкания. Если потребуется установить скалярный контекст для аргументов этих функций (что может быть не лучшим решением в данном примере), можно написать так:

```
*$name = sub ($) { "<FONT COLOR='$name'>$_[0]</FONT>" };
```

Почти хорошо. И все же, поскольку проверка прототипа выполняется на этапе компиляции, от присваивания на этапе выполнения мало пользы – оно происходит слишком поздно. Положение можно исправить, поместив весь цикл в блок BEGIN, чтобы присваивание происходило на этапе компиляции. (Более вероятно, что цикл придется поместить в модуль, загружаемый с помощью use на этапе компиляции.) В этом случае прототипы будут видимы на всем протяжении компиляции.

Вложенные подпрограммы

Если вы привыкли (в других языках) использовать подпрограммы, вложенные в другие подпрограммы и имеющие свои локальные переменные, вам придется приложить дополнительные усилия, чтобы использовать такой прием в Perl.

Именованные подпрограммы, в отличие от анонимных, не умеют играть в матрешки.[1] Но вложенные подпрограммы с лексической видимостью можно эмулировать посредством замыканий. Например:

```
sub outer {
    my $x = $_[0] + 35;
    local *inner = sub { return $x * 19 }
    return $x + inner();
}
```

Теперь inner может вызываться только из outer, поскольку в замыкании выполняются временные присваивания. Но в момент вызова inner получает обычный доступ к лексической переменной $x из области видимости outer.

При этом возникает интересный эффект создания функции, локальной для другой функции, что в Perl, вообще говоря, не поддерживается. Поскольку local имеет динамическую область видимости, а имена функций являются глобальными в своих пакетах, любая другая функция, которую вызовет outer, также сможет вызвать временную версию inner. Чтобы помешать этому, нужно ввести еще один уровень косвенной адресации:

```
sub outer {
    my $x = $_[0] + 35;
    my $inner = sub { return $x * 19 };
    return $x + $inner->();
}
```

Символические ссылки

Что произойдет, если попытаться разыменовать значение, не являющееся жесткой ссылкой? Это значение будет рассматриваться как *символическая ссылка* (*symbolic reference*). То есть ссылка будет интерпретироваться как строка с *именем* глобальной переменной.

Например:

```
$name = "bam";
$$name = 1;          # Установит $bam
$name->[0] = 4;      # Установит первый элемент @bam
$name->{X} = "Y";    # Установит элемент X хеша %bam равным Y
@$name = ();         # Очистит @bam
keys %$name;         # Вернет ключи %bam
&$name;              # Вызовет &bam
```

Это очень мощное средство и немного опасное, потому что можно намереваться (совершенно искренне) использовать жесткую ссылку, но случайно применить символическую. Чтобы защититься от этого, можно сделать так:

```
use strict "refs";
```

---

[1] Точнее, этого не умеют глобальные *именованные* подпрограммы. К сожалению, другого типа объявлений именованных подпрограмм у нас нет. Мы пока не реализовали именованные подпрограммы с лексической областью видимости (известные как my sub), но когда мы это сделаем, они будут поддерживать вложенность.

Теперь до конца охватывающего блока будут разрешены только жесткие ссылки. Вложенный блок сможет отменить это решение с помощью

```
no strict "refs";
```

Важно также понимать различия между следующими двумя строками кода:

```
${identifier},   # То же, что $identifier.
${"identifier"}; # Тоже $identifier, но символическая ссылка.
```

Поскольку во второй форме использованы двойные кавычки, она будет считаться символической ссылкой и вызовет сообщение об ошибке в области действия `use strict "refs"`. Даже если `strict "refs"` не действует, такая форма может ссылаться только на переменную пакета. Первая форма идентична форме без скобок и сможет ссылаться даже на переменную с лексической областью видимости, если таковая объявлена. Эта возможность демонстрируется в следующем примере (и обсуждается в следующем разделе).

Через символические ссылки доступны только переменные пакета, потому что поиск символических ссылок всегда выполняется в таблице имен пакета. Поскольку лексические переменные не включаются в таблицу имен пакета, они невидимы для данного механизма. Например:

```
our $value = "global";
{
    my $value = "private";
    print "Inside, mine is ${value}, ";
    say "but ours is ${"value"}.";
}
say "Outside, ${value} is again ${"value"}.";
```

в результате будет выведено:

```
Inside, mine is private, but ours is global.
Outside, global is again global.
[Внутри - мое приватное, а наше - глобальное.
Снаружи глобальное так и останется глобальным.]
```

Фигурные скобки, квадратные скобки и кавычки

В предыдущем разделе отмечалось, что `${identifier}` не считается символической ссылкой. Вас может заинтересовать, как в этом участвуют зарезервированные слова, и кратким ответом будет: «Никак». Хотя `push` является зарезервированным словом, следующие две инструкции выведут `"pop on over"`:

```
$push = "pop on ";
print "${push}over";
```

Причина в том, что исторически таким способом в оболочках UNIX имена переменных отделяются от последующего буквенно-цифрового текста, который, в противном случае, интерпретировался бы как часть имени. Именно такого поведения многие ожидают от интерполяции переменных, поэтому Perl работает так же. Но в Perl эта идея получила дальнейшее развитие и применяется к любым фигурным скобкам, используемым при создании ссылок, независимо от того, заключены ли они в кавычки. Это значит, что:

```
print ${push}   'over"
```

или даже (поскольку пробелы никогда не учитываются):

```
print ${ push }   "over",
```

выведет "pop on over', несмотря на то, что фигурные скобки находятся вне двойных кавычек. То же правило действует для любого идентификатора, используемого в качестве индекса хеша. Поэтому вместо:

```
$hash{ "aaa" }{ 'bbb" }{ "ccc" }
```

можно написать просто:

```
$hash{ aaa }{ bbb }{ ccc }
```

или:

```
$hash{aaa}{bbb}{ccc}
```

и не беспокоиться, что индексы могут оказаться зарезервированными словами. Поэтому выражение:

```
$hash{ shift }
```

интерпретируется как `$hash{"shift"}`. Добиться интерпретации индекса в качестве зарезервированного слова можно, добавив что-то, что выведет текст в скобках за рамки синтаксиса идентификатора:

```
$hash{  shift() }
$hash{ +shift }
$hash{  shift @_ }
```

Ссылки не могут быть ключами хеша

Ключи хешей внутренне хранятся как строки.[1] Если попытаться использовать ссылку в качестве ключа хеша, значение ключа будет преобразовано в строку:

```
$x{ \$a } = $a;
($key, $value) = each %x,
print $$key;                  # НЕВЕРНО
```

Выше говорилось, что нельзя преобразовать строку обратно в жесткую ссылку. Поэтому, если попытаться разыменовать `$key`, содержащую простую строку, произойдет разыменование не жесткой ссылки, а символической – а так как весьма вероятно, что переменной с именем `SCALAR(0x1fc0e)` нет, попытка не даст желаемого результата. Скорее, вам требуется что-то вроде этого:

```
$r = \@a;
$x{ $r } = $r
```

Тогда, по крайней мере, можно будет использовать *значение* хеша, которое будет жесткой ссылкой, а не ключ, который ею не будет.

Хоть и нельзя использовать ссылку как ключ, если жесткая ссылка используется в строковом контексте (как было в примере выше), она *гарантированно* соз-

---

[1] Они и *внешне* хранятся как строки, например, их помещают в файл DBM. На самом деле, файлы DBM *требуют*, чтобы их ключи (и значения) были строками.

даст уникальную строку, поскольку в нее будет включен адрес ссылки. Поэтому на практике ссылку можно использовать в качестве уникального ключа хеша. Просто потом нельзя будет разыменовать ее.

Существует один особый вид хеша, в котором можно использовать ссылки в качестве ключей. Благодаря волшебству (*magic*)[1] модуля Tie::RefHash, поставляемого в составе Perl, невозможное становится возможным:

```
use Tie::RefHash;
tie my %h, "Tie::RefHash";
%h = (
    ["this", "here"]  => "at home",
    ["that", "there"] => "elsewhere",
);
while ( my($keyref, $value) = each %h ) {
    say "@$keyref is $value";
}
```

На самом деле, связывая (tie) различные реализации со встроенными типами, можно добиться от скаляров, хешей и массивов многого из того, чего, как мы утверждаем, они делать не могут. Это послужит нам уроком! Мы такие бестолковые...

Подробнее о связывании – в главе 14.

Сборка мусора, циклические ссылки и слабые ссылки

Языки высокого уровня обычно позволяют программистам не заботиться об освобождении памяти, которая больше не используется. Этот автоматический процесс переработки отходов известен как *сборка мусора* (*garbage collection*). В большинстве случаев Perl использует быстрый и простой механизм сборки мусора, основанный на ссылках.

При выходе из блока его переменные с локальной областью видимости обычно освобождаются, однако можно так спрятать свой мусор, что сборщик мусора его не найдет. Одна из серьезных опасностей заключается в том, что недоступная память с ненулевым счетчиком ссылок обычно не может быть освобождена. Поэтому применение циклических ссылок – плохая идея:

```
{               # заставим $a и $b указывать друг на друга
    my ($a, $b);
    $a = \$b;
    $b = \$a;
}
```

или еще проще:

```
{               # заставим $a указывать на саму себя
    my $a;
    $a = \$a;
}
```

Переменную $a следует освободить в конце блока, однако этому не бывать. В рекурсивных структурах данных необходимо самостоятельно разрывать (или ослаб-

[1] Да, это *действительно* технический термин, как можно заметить, если просмотреть файл *mg.c* в дистрибутиве исходного кода Perl.

лять, см. ниже) автоссылки, чтобы память была утилизирована прежде, чем программа (или поток) завершит работу. (При выходе память будет освобождена автоматически путем дорогостоящей, но полной процедуры сборки мусора.) Если структура данных является объектом, для автоматического разрыва ссылки можно применить метод DESTROY; см. раздел «Сборка мусора с помощью методов DESTROY» в главе 12.

Аналогичная ситуация может иметь место с *кэшами* – хранилищами данных, предназначенными для ускорения доступа к этим данным. Вне кэша могут существовать ссылки на данные, находящиеся в нем. Проблема возникает, когда все эти ссылки уничтожаются, а данные в кэше со своими внутренними ссылками продолжают существовать. Существование хотя бы одной ссылки не позволяет утилизировать объект ссылки, даже если мы хотим, чтобы данные исчезали из кэша, как только они становятся не нужны. Как и в случае циклических ссылок, нам нужна ссылка, которая не влияет на счетчик ссылок и не задерживает сборку мусора.

Ниже демонстрируется еще один пример, на этот раз – двусвязного циклического списка:

```
$ring = {
    VALUE => undef,
    NEXT  => undef,
    PREV  => undef,
};
$ring->{NEXT} = $ring;
$ring->{PREV} = $ring;
```

Счетчик ссылок на лежащий в основе хеш равен трем, а операция присваивания значения undef переменной $ring или выход ее за пределы области видимости уменьшит счетчик только на единицу: в результате память, занимаемая хешем, не может быть автоматически освобождена сборщиком мусора Perl.

Для исправления этой ситуации в Perl было введено понятие *слабых ссылок* (*weak references*). Слабая ссылка действует подобно любым другим ссылкам (имеются в виду «жесткие», а не «символические» ссылки), за исключением двух важных особенностей: она не участвует в вычислении значения счетчика ссылок на объект ссылки, а когда объект ссылки утилизируется сборщиком мусора, ссылка автоматически получает значение undef. Благодаря этим особенностям слабые ссылки отлично подходят для реализации структур данных, хранящих внутренние ссылки. Внутренние ссылки не будут учитываться счетчиком ссылок, а внешние будут.

Хотя Perl поддерживал слабые ссылки начиная с версии v5.6, стандартная функция weaken для доступа к ним из программ на Perl появилась только в Perl версии v5.8.1, в составе стандартного модуля Scalar::Util. В этом модуле имеется также функция is_weak, позволяющая определить, является ли указанная ссылка слабой.

Ниже показано, как можно было бы переписать реализацию циклического списка:

```
use Scalar::Util qw(weaken);

$ring = {
    VALUE => undef,
    NEXT  => undef,
```

```
    PREV  => undef,
};
$ring->{NEXT} = $ring;
$ring->{PREV} = $ring;
weaken($ring->{NEXT});
weaken($ring->{PREV})
```

С точки зрения оператора ref, слабые ссылки действуют подобно обычным (жестким) ссылкам: они позволяют определить тип объекта ссылки. Но когда объект слабой ссылки утилизируется сборщиком мусора, переменная, хранящая слабую ссылку, автоматически получает неопределенное значение, так как не может ссылаться на несуществующий объект.

Операция копирования слабой ссылки создает обычную ссылку. Если в результате копирования необходимо получить еще одну слабую ссылку, полученную копию следует передать функции weaken.

Более объемный пример управления слабыми ссылками приводится в книге «Perl Cookbook», в рецепте 11.15 «Coping with Circular Data Structures using Weak References» (копирование циклических структур данных, использующих слабые ссылки) (*http://my.safaribooksonline.com/book/programming/perl/0596003137/references-and-records/perlckbk2-chp-11-sect-15*).

9

Структуры данных

Perl предлагает множество готовых структур данных, которые в других языках приходится создавать самостоятельно. Стеки и очереди, изучаемые будущими специалистами по информатике, в Perl являются простыми массивами. Если к массиву применяются функции `push` и `pop` (или `shift` и `unshift`), он ведет себя как стек; а если `push` и `shift` (или `unshift` и `pop`), он ведет себя как очередь. А многие древовидные структуры создаются только для ускорения доступа к концептуально плоским таблицам поиска. В Perl, разумеется, есть встроенные хеши для быстрого доступа к концептуально плоским таблицам поиска без использования структур данных, от рекурсивности которых цепенеет мозг, и кажущихся красивыми лишь тем, чей мозг уже достаточно оцепенел.

Но иногда вложенные структуры данных действительно нужны, поскольку позволяют более естественно моделировать решаемую задачу. Поэтому Perl позволяет комбинировать и вкладывать массивы и хеши для создания структур данных произвольной сложности. При правильном использовании они могут применяться для создания связанных списков, двоичных деревьев, куч, B-деревьев, множеств, графов и всего, что вы еще придумаете. См. книги «Mastering Algorithms with Perl», O'Reilly, 1999 (Язык Perl и реализация алгоритмов); «Perl Cookbook», O'Reilly, 1998[1]; раздел «Data Structure Cookbook» в страницах *perldsc* справочного руководства или CPAN – главное хранилище таких модулей. Но возможно, что вам никогда не понадобится ничего, кроме простых сочетаний массивов и хешей, поэтому о них-то мы и поговорим в этой главе.

Массивы массивов

Существует много видов вложенных структур данных. Проще всего построить массив массивов, называемый также двумерным массивом или матрицей. (Допустимо очевидное обобщение: массив массивов массивов является трехмерным

---

1 Т. Кристиансен, Н. Торкингтон «Perl: библиотека программиста». – Пер. с англ. – СПб.: Питер, 2001.

массивом и так далее для большего числа измерений.) Сам принцип достаточно прост, и почти все, что применимо в данном случае, может применяться к более замысловатым структурам данных, которые мы будем изучать в следующих разделах.

Создание и использование двумерного массива

Вот как создается двумерный массив:

```
# Присвоить массиву список ссылок на массивы.
@AoA = (
        [ "fred", "barney" ],
        [ "george", "jane", "elroy" ],
        [ "homer", "marge", "bart" ],
);

print $AoA[2][1]; # выведет "marge"
```

Весь список заключен в круглые, а не квадратные скобки, потому что присваивается списочное значение, а не ссылка. Чтобы получить ссылку на массив, следует использовать квадратные скобки:

```
# Создать ссылку на массив ссылок на массивы.
$ref_to_AoA = [
    [ "fred", "barney", "pebbles", "bamm bamm", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];

print $ref_to_AoA->[2][3]; # выведет "judy"
```

Помните, что между каждой парой смежных фигурных или квадратных скобок подразумевается присутствие ->. Поэтому эти две строки:

```
$AoA[2][3]
$ref_to_AoA->[2][3]
```

эквивалентны следующим двум строкам:

```
$AoA[2]->[3]
$ref_to_AoA->[2]->[3]
```

Но здесь не подразумевается присутствие -> перед первой парой квадратных скобок, поэтому для разыменования `$ref_to_AoA` необходима первая стрелка ->. Помните также, что с помощью отрицательных индексов можно вести обратный отсчет от конца массива, поэтому:

```
$AoA[0][-2]
```

вернет предпоследний элемент первой строки.

Наращивание массива

Присваивание больших списков годится только для создания фиксированных структур данных, но как быть, если требуется конструировать структуру по частям или вычислять ее элементы «на лету»?

Прочтем структуру данных из файла. Предположим, что это простой текстовый файл, где каждая строка содержит структурированные данные и состоит из элементов, разделяемых пробельными символами. Вот как нужно действовать:[1]

```
while (<>) {
    @tmp = split;        # Расщепить элементы в массив.
    push @AoA, [ @tmp ]; # Добавить в @AoA ссылку на анонимный массив.
}
```

Конечно, нет необходимости давать имя временному массиву, поэтому можно сказать и так:

```
while (<>) {
    push @AoA, [ split ];
}
```

Получить ссылку на массив массивов можно так:

```
while (<>) {
    push @$ref_to_AoA, [ split ];
}
```

В обоих примерах к массиву массивов добавляются новые строки. А как добавить новые колонки? При работе с двумерными массивами часто проще использовать обычное присваивание:[2]

```
for $x (0 .. 9) {                       # Для каждой строки..
    for $y (0 .. 9) {                   # Для каждой колонки...
        $AoA[$x][$y] = func($x, $y);    # ...присвоить этой ячейке
    }
}

for $x ( 0..9 ) {                       # Для каждой строки...
    $ref_to_AoA->[$x][3] = func2($x);   # ...присвоить четвертой колонке
}
```

Порядок присваивания значений элементам не важен, как и наличие в @AoA элементов с указанными индексами: Perl охотно создаст их для вас, установив неопределенное значение в промежуточных элементах (при необходимости Perl даже создаст первую ссылку в $ref_to_AoA). Чтобы просто добавить элементы в конец строки, нужно проявить немного смекалки:

```
# Добавить новые колонки к существующей строке
push @{ $AoA[0] }, "wilma", "betty";
```

У кого-то, возможно, возникнет вопрос: можно ли избежать разыменования и просто записать:

```
push $AoA[0], "wilma", "betty"; # ошибка компиляции < v5.14!
```

Мы тоже задались тем же вопросом. Долгое время этот код вообще не компилировался, потому что аргумент функции push должен быть настоящим массивом, а не

---

1 Здесь, как и в других главах, мы опускаем (для ясности) объявления my. В данном примере надо было бы написать my @tmp = split.

2 Как и в присваивании временной переменной выше, мы упростили код: в рабочем коде для циклов, обсуждаемых в этой главе, эта форма выглядела бы как for my $x.

ссылкой на массив. Поэтому первый аргумент обязательно должен начинаться символом @. То, что следует за @, может быть предметом обсуждения.

Начиная с версии v5.14 *иногда* можно опускать явное разыменование при вызове некоторых встроенных функций. В число этих функций входят: pop, push, shift, unshift и splice – для массивов, и keys, values, each – для хешей. Они больше не требуют, чтобы первый аргумент начинался с символа @ или %. Если функции передать допустимую ссылку на коллекцию соответствующего типа, она будет разыменована при необходимости; в отличие от явного разыменования, такое неявное разыменование никогда не приводит к автовивификации. Если передать недопустимую ссылку, во время выполнения возникнет исключение. Поскольку при компиляции подобного кода, использующего новые возможности, старый компилятор обязательно подавится, вы должны предупредить пользователей об этом, поместив прагму use *VERSION* в начало файла:

```
use 5.014,      # запретить розлив молодого вина в старые бутылки
use v5.14;      # запретить накладывать новые заплаты на старые платья
```

Доступ к массивам и вывод

А теперь выведем структуру данных. Если вас интересует лишь один элемент, достаточно следующего:

```
print $AoA[3][2];
```

Но чтобы вывести весь массив, нельзя просто сказать:

```
print @AoA;     # НЕВЕРНО
```

Это неверное решение, поскольку вместо значений вы увидите ссылки, преобразованные в строки. Perl не выполняет разыменование автоматически, поэтому придется написать один-два цикла. Следующий код выведет всю структуру, выполнив обход элементов @AoA в цикле с разыменованием каждого из них в отдельности в операторе say:

```
for $row ( @AoA ) {
    say "@$row";
}
```

Если требуется следить за индексами, можно сделать так:

```
for $i ( 0 .. $#AoA ) {
    say "строка $i: @{$AoA[$i]}"
}
```

или даже так (обратите внимание на внутренний цикл):

```
for $i ( 0 .. $#AoA ) {
    for $j ( 0 .. $#{$AoA[$i]} ) {
        say "элемент $i $j is $AoA[$i][$j]";
    }
}
```

Как видите, дело немного усложняется. Поэтому иногда проще решить задачу с помощью временной переменной:

```
for $i ( 0 .. $#AoA ) {
    $row = $AoA[$i];
```

```
        for $j ( 0 .. $#{$row} ) {
            say "элемент $i $j   $row->[$j]";
        }
    }
```

Утомившись созданием пользовательских функций вывода для своих структур данных, обратите внимание на стандартные модули Dumpvalue и Data::Dumper. Первый используется отладчиком Perl, а второй генерирует код, который может быть разобран анализатором Perl. Например:

```
use v5.14;             # используется прототип +, появился в v5.14

sub show(+) {
    require Dumpvalue;
    state $prettily = new Dumpvalue::
                        tick        => q("),
                        compactDump => 1, # закомментируйте эти две строки
                        veryCompact => 1, # чтобы получить более подробный вывод
                    ;
    dumpValue $prettily @_;
}

# Присвоить массиву список ссылок на массивы.
my @AoA = (
        [ "fred", "barney" ],
        [ "george", "jane", "elroy" ],
        [ "homer", "marge", "bart" ],
);
push $AoA[0], "wilma", "betty";
show @AoA;
```

выведет:

```
0 0..3 "fred" "barney" "wilma" "betty"
1 0..2 "george" "jane" "elroy"
2 0..2 "homer" "marge" "bart"
```

Если закомментировать две строки, упомянутые в тексте программы, содержимое массива будет выведено в таком виде:

```
0 ARRAY(0x8031d0)
  0 "fred"
  1 "barney"
  2 "wilma"
  3 "betty"
1 ARRAY(0x803d40)
  0 "george"
  1 "jane"
  2 "elroy"
2 ARRAY(0x803e10)
  0 "homer"
  1 "marge"
  2 "bart"
```

Нам симпатичен модуль Data::Dump из архива CPAN, который также можно использовать для вывода данных. Например, следующий код:

```
use v5.14;                    # для поддержки скаляров в функции push
use Data::Dump qw(dump);      # модуль из CPAN

my @AoA = (
        [ "fred", "barney" ],
        [ "george", "jane", "elroy" ],
        [ "homer", "marge", "bart" ],
);
push $AoA[0], "wilma", "betty";
dump \@AoA;
```

выведет:

```
[
  ["fred", "barney", "wilma", "betty"],
  ["george", "jane", "elroy"],
  ["homer", "marge", "bart"],
]
```

Срезы

Если нужно получить доступ к срезу (фрагменту строки) многомерного массива, придется использовать индексы несколько необычным способом. Стрелки указателей обеспечивают удобный доступ к отдельным элементам, но для срезов таких удобств не существует. Элементы среза можно извлечь последовательно с помощью цикла:

```
@part = ();
for ($y = 7; $y < 13; $y++) {
    push @part, $AoA[4][$y];
}
```

Данный конкретный цикл можно заменить срезом массива:

```
@part = @{ $AoA[4] } [ 7..12 ]
```

Если же требуется *двумерный срез*, например для `$x` в диапазоне `4..8` и `$y` в диапазоне `7..12`, его можно получить так:

```
@newAoA = ();
for ($startx = $x = 4; $x <= 8; $x++) {
    for ($starty = $y = 7; $y <= 12; $y++) {
        $newAoA[$x - $startx][$y - $starty] = $AoA[$x][$y];
    }
}
```

В этом примере отдельным элементам двумерного массива `@newAoA` последовательно присваиваются значения, извлеченные из двумерного подмассива `@AoA`. Как вариант можно создать анонимные массивы, каждый из которых содержит требуемый срез подмассива `@AoA`, и поместить ссылки на эти анонимные массивы в `@newAoA`. В этом случае мы будем записывать ссылки в `@newAoA` (используя, если можно так выразиться, один индекс), а не значения из подмассива, используя два индекса. При таком подходе устраняется самый цикл самой глубокой вложенности:

```
for ($x = 4; $x <= 8; $x++) {
    push @newAoA, [ @{ $AoA[$x] } [ 7..12 ] ];
}
```

Конечно, если вы делаете это часто, вероятно, следует написать подпрограмму, назвав ее, например, extract_rectangle. А если редко и для больших наборов многомерных данных, то следует, вероятно, воспользоваться модулем PDL (Perl Data Language), который можно получить из CPAN.

Распространенные ошибки

Как говорилось выше, массивы и хеши в Perl являются одномерными. Даже «многомерные» массивы фактически имеют одно измерение, но значения в них являются ссылками на другие массивы, «схлопывающими» несколько элементов в один. При выводе этих значений без разыменования вы увидите не значения, а ссылки, преобразованные в строки. Например, следующие две строки:

```
@AoA = ( [2, 3], [4, 5, 7], [0] );
print "@AoA";
```

выведут что-нибудь вроде:

```
ARRAY(0x83c38) ARRAY(0x8b194) ARRAY(0x8b1d0)
```

С другой стороны, следующая строка выведет 7:

```
print $AoA[1][2];
```

Конструируя массивы массивов, не забывайте создавать новые ссылки для подмассивов, иначе получится массив, элементы которого будут представлять число элементов в подмассивах, например:

```
for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = @array;      # НЕВЕРНО!
}
```

Здесь доступ к @array осуществляется в скалярном контексте, поэтому возвращается число его элементов, которое Perl послушно присваивает элементу $AoA[$i]. Как правильно присваивать ссылку, будет показано чуть ниже.

Допустив такую ошибку, человек понимает, что нужно присвоить ссылку, поэтому следующая естественная ошибка – это многократное получение ссылки на один и тот же адрес:

```
for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = \@array;     # СНОВА НЕВЕРНО!
}
```

Все ссылки, созданные во второй строке цикла, будут одинаковыми, а именно – все они будут ссылками на один и тот же массив @array. Да, этот массив изменяется в каждом цикле, но когда все закончится, $AoA будет содержать 10 ссылок на один и тот же массив, содержащий последний набор присвоенных ему значений. Вызов print @{$AoA[1]} выведет те же значения, что и print @{$AoA[2]}.

Вот более удачное решение:

```
for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = [ @array ]; # ПРАВИЛЬНО!
}
```

Квадратные скобки вокруг массива @array создают новый анонимный массив, куда копируются значения элементов @array. Затем мы сохраняем ссылку на этот новый массив.

Следующий (более трудный для чтения) код даст аналогичный результат:

```
for $i (1..10) {
    @array = somefunc($i);
    @{$AoA[$i]} = @array;
}
```

Поскольку элемент $AoA[$i] должен быть новой ссылкой, она будет создана автоматически. Затем предшествующий символ @ разыменует новую ссылку, в результате чего значения @array будут присвоены (в списочном контексте) массиву, на который ссылается $AoA[$i]. Чтобы код был более понятным, такой конструкции стоит избегать.

Но в одном случае ее все же можно использовать. Допустим, что @AoA уже является массивом ссылок на массивы, т.е. выполнены присваивания вида:

```
$AoA[3] = \@original_array;
```

А теперь предположим, что нужно изменить @original_array (т.е. четвертую строку $AoA) так, чтобы он ссылался на элементы @array. Сделать это можно так:

```
@{$AoA[3]} = @array;
```

В данном случае сама ссылка не меняется, изменяются элементы массива, на который она ссылается. В результате перезаписываются значения в массиве @original_array.

Наконец, следующий код прекрасно работает, хотя и выглядит жутковато:

```
for $i (1..10) {
    my @array = somefunc($i);
    $AoA[$i] = \@array;
}
```

Это объясняется тем, что переменная с лексической областью видимости my @array создается заново в каждой итерации цикла. И хотя кажется, будто каждый раз сохраняется ссылка на одну и ту же переменную, на самом деле это не так. Это тонкое отличие создает более эффективный код, хотя при этом возникает опасность ввести в заблуждение не слишком опытных программистов. (Он более эффективен, потому что в последнем присваивании не выполняется копирование.) С другой стороны, если значения все равно приходится копировать (в первом присваивании в цикле), можно использовать копирование, выполняемое квадратными скобками, и избавиться от временной переменной:

```
for $i (1..10) {
    $AoA[$i] = [ somefunc($i) ];
}
```

Подведем итоги:

```
$AoA[$i] = [ @array ];      # Самое безопасное, иногда самое быстрое
$AoA[$i] = \@array;         # Быстро, но рискованно,
                            # зависит от наличия my при массиве
@{ $AoA[$i] } = @array;     # Несколько мудрено
```

Овладев многомерными массивами, вы сможете заняться более сложными структурами данных. Вы не найдете в Perl зарезервированных слов для создания структур С или записей Pascal. Вместо них предоставляется более гибкая система. Если вы считаете, что структура записи должна быть менее гибкой или хотите дать своим пользователям нечто менее прозрачное и более жесткое, используйте возможности объектно-ориентированного программирования, обсуждаемые в главе 12.

В Perl всего два способа организации данных: упорядоченные списки, хранимые в массивах, с доступом по индексам, и неупорядоченные пары ключ/значение, хранимые в хешах, с доступом по именам. Для представления записей в Perl лучше всего подходят ссылки на хеш, но выбор способа организации таких записей может быть различным. Кому-то понадобится хранить упорядоченный список этих записей, и тогда следует использовать массив ссылок на хеши. Кому-то может потребоваться поиск записей по именам, и тогда следует создать хеш ссылок на хеши.

В следующих разделах приводятся примеры, подробно показывающие, как создавать (с нуля), генерировать (на основе других источников данных), использовать и отображать структуры данных нескольких различных типов. Сначала мы покажем три простые комбинации массивов и хешей, затем хеш функций и более необычные структуры данных. И в заключение научим сохранять эти структуры данных. Приведенные примеры предполагают, что вы хорошо освоили материал, изложенный ранее в этой главе.

Хеши массивов

Используйте хеш массивов, когда нужно находить каждый массив по некоторой строке, вместо номера индекса. В нашем примере с персонажами телесериалов вместо того, чтобы просматривать списки имен 0-й серии, 1-й серии и т. д., мы сделаем так, что можно будет найти список действующих лиц по названию серии.

Поскольку внешней структурой данных является хеш, мы не можем упорядочить содержимое, но можем использовать функцию sort для вывода данных в определенном порядке.

Формирование хеша массивов

Хеш анонимных массивов можно создать так:

```
# Обычно мы опускаем кавычки, когда ключи являются идентификаторами
%HoA = (
    flintstones => [ "fred", "barney" ],
    jetsons     => [ "george", "jane", "elroy" ],
    simpsons    => [ "homer", "marge", "bart" ],
);
```

Чтобы добавить в хеш еще один массив, можно просто сказать:

```
$HoA{teletubbies} = [ "tinky winky", "dipsy", "laa-laa", "po" ];
```

Генерирование хеша массивов

Вот несколько приемов заполнения хешей массивов. Чтобы произвести чтение из файла следующего формата:

```
flintstones: fred barney wilma dino
jetsons:     george jane elroy
simpsons:    homer marge bart
```

можно использовать любой из двух циклов:

```
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $HoA{$1} = [ split ];
}

while ( $line = <> ) {
    ($who, $rest) = split /:\s*/, $line, 2;
    @fields = split " ", $rest;
    $HoA{$who} = [ @fields ];
}
```

Если имеется подпрограмма get_family**, возвращающая массив, ее можно использовать для заполнения** %HoA **с помощью любого из следующих циклов:**

```
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoA{$group} = [ get_family($group) ];
}

for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoA{$group} = [ @members ];
}
```

Новые члены можно дописать в существующий массив следующим образом:

```
push @{ $HoA{flintstones} }, "wilma", "pebbles";
```

Доступ к хешу массивов и вывод его элементов

Присвоить значение первому элементу некоторого массива можно так:

```
$HoA{flintstones}[0] = "Fred";
```

Чтобы преобразовать первый символ в имени второго Симпсона в верхний регистр, примените операцию подстановки к соответствующему элементу массива:

```
$HoA{simpsons}[1] =~ s/(\w)/\u$1/;
```

Вывести членов всех семейств можно, выполнив обход всех ключей хеша в цикле:

```
for $family ( keys %HoA ) {
    say "$family: @{ $HoA{$family} }";
}
```

Приложив дополнительные усилия, можно добавить также индексы массивов:

```
for $family ( keys %HoA ) {
    print "$family: ";
    for $i ( 0 .. $#{ $HoA{$family} } ) {
        print " $i = $HoA{$family}[$i]";
    }
    print "\n";
}
```

Или отсортировать массивы по числу элементов в них:

```
for $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {
    say "$family: @{ $HoA{$family} }"
}
```

Или даже отсортировать массивы по числу элементов, а затем упорядочить элементы в алфавитном порядке, в соответствии с таблицей ASCII (точнее, utf8):

```
# Вывести все, упорядочив по числу элементов и именам.
for $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {
    say "$family: ", join(", ", sort @{ $HoA{$family} });
}
```

Если фамилии содержат не-ASCII символы Юникода или какую-либо пунктуацию, сортировка по кодам символов не позволит получить алфавитный порядок. Вместо этого используйте следующий прием:

```
use Unicode::Collate;
my $sorter = Unicode::Collate->new(); # обычная алфавитная сортировка
say "$family: ",
    join ", " => $sorter->sort( @{ $HoA{$family} } );
```

Массивы хешей

Массив хешей удобен, когда есть группа записей, для которых обращение осуществляется последовательно, и каждая запись содержит пары ключ/значение. Массивы хешей используются реже, чем прочие структуры, представленные в данной главе.

Формирование массива хешей

Создать массив анонимных хешей можно так:

```
@AoH = (
    {
        husband => "barney",
        wife    => "betty",
        son     => "bamm bamm",
    },
    {
        husband => "george",
        wife    => "jane",
        son     => "elroy",
    },
    {
        husband => "homer",
        wife    => "marge",
        son     => "bart",
    },
);
```

Чтобы добавить в массив еще один хеш, можно просто сказать:

```
push @AoH, { husband => "fred", wife => "wilma", daughter => "pebbles" };
```

Генерирование массива хешей

Вот несколько приемов заполнения массива хешей. Чтобы прочитать массив из файла следующего формата:

```
husband=fred friend=barney
```

можно использовать один из двух следующих циклов:

```
while ( <> ) {
    $rec = {};
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
    push @AoH, $rec;
}

while ( <> ) {
    push @AoH, { split /[\s=]+/ };
}
```

Если имеется подпрограмма `get_next_pair`, возвращающая пары ключ/значение, ее можно применить для заполнения `@AoH` с помощью любого из следующих циклов:

```
while ( @fields = get_next_pair() ) {
    push @AoH, { @fields };
}

while (<>) {
    push @AoH, { get_next_pair($_) };
}
```

Новые члены можно дописать в существующий хеш так:

```
$AoH[0]{pet} = "dino";
$AoH[2]{pet} = "santa's little helper";
```

Доступ к массиву хешей и вывод его элементов

Установить пару ключ/значение в некотором хеше можно так:

```
$AoH[0]{husband} = "fred";
```

Чтобы преобразовать в верхний регистр первый символ в значении ключа `husband` из второго массива, примените операцию подстановки:

```
$AoH[1]{husband} =~ s/(\w)/\u$1/;
```

Вывести все данные можно так:

```
for $href ( @AoH ) {
    print "{ ";
    for $role ( keys %$href ) {
        print "$role=$href->{$role} ";
    }
    print "}\n";
}
```

А вот так их можно вывести с индексами:

```
for $i ( 0 .. $#AoH ) {
    print "$i is { ";
    for $role ( keys %{ $AoH[$i] } ) {
        print "$role=$AoH[$i]{$role} ";
    }
    print "}\n";
}
```

Хеши хешей

Многомерный хеш является самой гибкой из вложенных структур Perl. Он напоминает запись, содержащую другие записи. На каждом уровне хеш индексируется строкой (в кавычках, если необходимо). Но не забывайте, что пары ключ/значение извлекаются из хеша в произвольном порядке. Для извлечения пар в нужном порядке можно использовать функцию sort.

Формирование хеша хешей

Хеш анонимных хешей можно создать так:

```
%HoH = (
    flintstones => {
        husband   => "fred",
        pal       => "barney",
    },
    jetsons => {
        husband   => "george",
        wife      => "jane",
        "his boy" => "elroy",  # Этот ключ должен быть закавычен
    },
    simpsons => {
        husband   => "homer",
        wife      => "marge",
        kid       => "bart",
    },
);
```

Чтобы добавить в %HoH еще один анонимный хеш, можно просто сказать:

```
$HoH{ mash } = {
    captain  => "pierce",
    major    => "burns",
    corporal => "radar",
};
```

Генерирование хеша хешей

Вот несколько приемов заполнения хеша хешей. Чтобы прочитать содержимое файла такого формата:

```
flintstones: husband=fred pal=barney wife=wilma pet=dino
```

можно использовать один из следующих циклов:

```
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $HoH{$who}{$key} = $value;
    }
}

while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    $rec = {};
    $HoH{$who} = $rec;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
}
```

Если имеется подпрограмма get_family, возвращающая список пар ключ/значение, ее можно использовать для заполнения %HoH с помощью одного из трех следующих фрагментов:

```
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoH{$group} = { get_family($group) };
}

for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoH{$group} = { @members };
}

sub hash_families {
    my @ret;
    for $group ( @_ ) {
        push @ret, $group, { get_family($group) };
    }
    return @ret;
}
%HoH = hash_families( "simpsons", "jetsons", "flintstones" );
```

Добавить новые члены в существующий хеш можно так:

```
%new_folks = (
    wife => "wilma",
    pet  => "dino";
);
for $what (keys %new_folks) {
    $HoH{flintstones}{$what} = $new_folks{$what};
}
```

Доступ к хешу хешей и вывод его элементов

Установить значение пары ключ/значение в хеше можно так:

```
$HoH{flintstones}{wife} = "wilma";
```

Чтобы преобразовать в верхний регистр первый символ в значении некоторого ключа, примените к соответствующему элементу операцию подстановки:

```
$HoH{jetsons}{'his boy'} =~ s/(\w)/\u$1/;
```

Вывести членов всех семейств можно, выполнив в цикле обход ключей внешнего хеша, а затем ключей внутреннего хеша:

```
for $family ( keys %HoH ) {
    print "$family: ";
    for $role ( keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "\n";
}
```

При работе с очень большими хешами немного быстрее будет одновременно извлекать ключи и значения с помощью each **(но это помешает сортировке):**

```
while ( ($family, $roles) = each %HoH ) {
    print "$family: ";
    while ( ($role, $person) = each %$roles ) {
        print "$role=$person ";
    }
    print "\n";
}
```

(К несчастью, именно большие хеши чаще всего приходится сортировать, в противном случае в выводимых результатах невозможно будет найти нужную информацию.) Отсортировать по семействам, а затем по ролям можно следующим образом:

```
for $family ( sort keys %HoH ) {
    print "$family: ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "\n";
}
```

Чтобы отсортировать семейства по числу членов (а не по кодам ASCII или utf8), можно использовать keys **в скалярном контексте:**

```
for $family ( sort { keys %{$HoH{$a}} <=> keys %{$HoH{$b}} } keys %HoH ) {
    print "$family: ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "\n";
}
```

Чтобы отсортировать членов семейств в некотором фиксированном порядке, можно присвоить им ранги:

```
$i = 0;
for ( qw(husband wife son daughter pal pet) ) { $rank{$_} = ++$i }

for $family ( sort { keys %{$HoH{$a}} <=> keys %{$HoH{$b}} } keys %HoH ) {
```

```
    print "$family: ";
    for $role ( sort { $rank{$a} <=> $rank{$b} } keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "\n";
}
```

Хеши функций

При создании сложного приложения или сетевой службы на языке Perl может возникнуть необходимость предложить пользователям значительное число команд. В такой программе для определения выбора пользователя и выполнения соответствующих действий может содержаться код вроде следующего:

```
if    ($cmd =~ /^exit$/i)   { exit }
elsif ($cmd =~ /^help$/i)   { show_help() }
elsif ($cmd =~ /^watch$/i)  { $watch = 1 }
elsif ($cmd =~ /^mail$/i)   { mail_msg($msg) }
elsif ($cmd =~ /^edit$/i)   { $edited++; editmsg($msg); }
elsif ($cmd =~ /^delete$/i) { confirm_kill() }
else {
    warn "Неизвестная команда: `$cmd'; попробуйте вызвать `help'\n";
}
```

Структуры данных способны хранить ссылки на функции точно так же, как ссылки на массивы и хеши:

```
%HoF = (          # Формирование хеша функций
    exit   => sub { exit },
    help   => \&show_help,
    watch  => sub { $watch = 1 },
    mail   => sub { mail_msg($msg) },
    edit   => sub { $edited++; editmsg($msg); },
    delete => \&confirm_kill,
);

if ($HoF{lc $cmd}) { $HoF{lc $cmd}->() }    # Вызов функции
else { warn "Неизвестная команда: `$cmd'; попробуйте вызвать `help'\n" }
```

В предпоследней строке проверяется, существует ли указанное имя команды (в нижнем регистре) в «таблице ссылок», %HoF. Если да, выполняется соответствующая команда за счет разыменования значения хеша как функции и передачи этой функции пустого списка аргументов. Разыменование можно было бы выполнить формой &{ $HoF{lc $cmd} }() или, в версии Perl 5.6, просто $HoF{lc $cmd}().

Более сложные записи

До сих пор в этой главе мы имели дело только с простыми двухуровневыми однородными структурами данных, где каждый элемент содержит объект ссылки того же типа, что и остальные элементы этого уровня. Разумеется, это не является обязательным требованием. Любой элемент может содержать скаляр любого типа, т.е. может быть строкой, числом или ссылкой на что угодно. Ссылка может ссылаться на массив или хеш, псевдохеш, именованную или анонимную функцию или

объект. Единственное, чего нельзя сделать, это поместить в один скаляр несколько объектов ссылки. Если вы поймали себя на попытке это сделать, значит, вам нужна ссылка на массив или хеш, чтобы объединить несколько значений в одно.

В последующих разделах вы найдете примеры, иллюстрирующие хранение многих возможных типов данных в записях, реализуемых с помощью ссылок на хеш. Ключами в них являются строки из символов верхнего регистра, – такое соглашение иногда применяется, когда хеш выступает в качестве особого типа записи.

Формирование, использование и вывод более сложных записей

Следующая запись содержит шесть полей совершенно разных типов:

```
$rec = {
    TEXT       => $string,
    SEQUENCE   => [ @old_values ],
    LOOKUP     => { %some_table },
    THATCODE   => \&some_function,
    THISCODE   => sub { $_[0] ** $_[1] },
    HANDLE     => \*STDOUT,
};
```

Поле *TEXT* является обыкновенной строкой, поэтому его можно вывести так:

```
print $rec->{TEXT};
```

SEQUENCE и *LOOKUP* – это обычные ссылки на массив и хеш:

```
print $rec->{SEQUENCE}[0];
$last = pop @{ $rec->{SEQUENCE} };

print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{ $rec->{LOOKUP} }
```

THATCODE – это именованная подпрограмма, а *THISCODE* – анонимная подпрограмма, но вызываются они одинаково:

```
$that_answer = $rec->{THATCODE}->($arg1, $arg2),
$this_answer = $rec->{THISCODE}->($arg1, $arg2);
```

С помощью дополнительной пары фигурных скобок можно обращаться с `$rec->{HANDLE}` как с косвенным объектом:

```
print { $rec->{HANDLE} } "строка\n";
```

А с помощью модуля `IO::Handle` с дескриптором можно обращаться как с обычным объектом:

```
use IO::Handle;
$rec->{HANDLE}->autoflush(1);
$rec->{HANDLE}->print("строка\n");
```

Формирование, использование и вывод еще более сложных записей

Естественно, что поля структур данных сами могут являться структурами данных произвольной сложности:

```
%TV = (
    flintstones => {
        series   => "flintstones",
        nights   => [ "monday" "thursday", "friday" ],
        members  => [
            { name => "fred",    role => "husband", age => 36, },
            { name => "wilma",   role => "wife",    age => 31, },
            { name => "pebbles", role => "kid",     age =>  4, },
        ],
    },

    jetsons    => {
        series   => "jetsons",
        nights   => [ "wednesday". "saturday" ]
        members  => [
            { name => "george", role => "husband", age => 41, },
            { name => "jane",   role => "wife",    age => 39, },
            { name => "elroy",  role => "kid",     age =>  9, }
        ],
    },

    simpsons   => {
        series   => "simpsons",
        nights   => [ "monday" ],
        members  => [
            { name => "homer", role => "husband", age => 34, },
            { name => "marge", role => "wife",    age => 37, },
            { name => "bart".  role => "kid".     age => 11, },
        ],
    },
);
```

Генерирование хеша сложных записей

Поскольку Perl хорошо справляется с анализом сложных структур данных, их можно поместить в отдельный файл в виде обычного кода на языке Perl, а затем загружать с помощью встроенных функций do **или** require**. Для загрузки произвольных структур данных, описанных на каком-нибудь другом языке (вроде XML), часто используются модули из CPAN (например,** XML::Parser**).**

Структуры данных можно строить по частям:

```
$rec = {};
$rec->{series} = "flintstones",
$rec->{nights} = [ find_days() ]
```

Или читать их из файла (здесь предполагается использование формата поле=значение**):**

```
@members = (),
while (<>) {
    %fields = split /[\s=]+/;
    push @members, { %fields }
}
$rec->{members} = [ @members ];
```

Вкладывать в более крупные структуры, используя в качестве ключа одно из вложенных полей:

```
$TV{ $rec->{series} } = $rec;
```

Применение дополнительных полей с указателями помогает избежать дублирования данных. Например, в запись о человеке можно включить поле "kids" со ссылкой на массив, содержащий ссылки на записи с информацией о детях. Связав части структуры ссылками (в противовес копированию), можно избежать нарушения целостности данных в результате обновления лишь в одном месте из нескольких необходимых:

```
for $family (keys %TV) {
    my $rec = $TV{$family};    # временный указатель
    @kids = ();
    for $person ( @{$rec->{members}} ) {
        if ($person->{role} =~ /kid|son|daughter/) {
            push @kids, $person;
        }
    }
    # $rec и $TV{$family} указывают на одни и те же данные!
    $rec->{kids} = [ @kids ],
}
```

Присваивание `$rec->{kids} = [ @kids ]` **копирует содержимое массива – ссылки на некопируемые данные. Это значит, что, если увеличить возраст Барта:**

```
$TV{simpsons}{kids}[0]{age}++;         # увеличивает до 12
```

то, поскольку `$TV{simpsons}{kids}[0]` **и** `$TV{simpsons}{members}[2]` **указывают на одну и ту же анонимную хеш-таблицу, мы увидим следующий результат:**

```
print $TV{simpsons}{members}[2]{age};  # выведет 12
```

Теперь выведем всю структуру %TV:

```
for $family ( keys %TV ) {
    print "the $family";
    print " is on ", join (" and ", @{ $TV{$family}{nights} })
    print "its members are:\n",
    for $who ( @{ $TV{$family}{members} } ) {
        print " $who->{name} ($who->{role}), age $who->{age}\n
    }
    print "children:  ,
    print join (", ", map { $_->{name} } @{ $TV{$family}{kids} } )
    print "\n\n";
}
```

Сохранение структур данных

Если потребуется сохранить структуры данных для использования в дальнейшем другой программой, сделать это можно несколькими способами. Проще всего применить модуль Perl `Data::Dumper`**, преобразующий структуру данных (возможно, содержащую ссылки на саму себя) в строку, которую можно сохранить во внешней памяти и впоследствии реанимировать с помощью** `eval` **или** `do`**.**

```
use Data::Dumper;
$Data::Dumper::Purity = 1;       # поскольку %TV содержит ссылки на себя
open (FILE, "> tvinfo.perldata") || die "невозможно открыть tvinfo: $!";
print FILE Data::Dumper->Dump([\%TV], ['*TV']);
close(FILE)                      || die "невозможно закрыть tvinfo: $!";
```

Отдельная программа (или та же самая) может позже прочитать этот файл:

```
open (FILE, "< tvinfo.perldata") || die "невозможно открыть tvinfo: $!";
undef $/;                        # прочитать файл целиком
eval <FILE>;                     # воссоздать %TV
die "невозможно воссоздать данные tv из tvinfo.perldata: $@" if $@;
close(FILE)                      || die "невозможно закрыть tvinfo: $!";
print $TV{simpsons}{members}[2]{age};
```

или просто:

```
do "tvinfo.perldata"            || die "невозможно воссоздать tvinfo: $! $@";
print $TV{simpsons}{members}[2]{age};
```

Storable, другой стандартный модуль, записывает структуры данных, упаковывая их в двоичный формат, пригодный для быстрой обработки. Он также поддерживает автоматическую блокировку файла (при условии, что ваша система реализует функцию flock) и даже предлагает точки перехвата (hooks), чтобы объекты могли сами выполнять сериализацию. Ниже показано, как ту же структуру сохранить с помощью модуля Storable:

```
use Storable qw(lock_nstore);
lock_nstore(\%TV, "tvdata.storable");
```

А следующий фрагмент демонстрирует чтение структуры из файла в переменную. После выполнения этой операции в переменной $TV_ref окажется ссылка на извлеченный хеш:

```
use Storable qw(lock_retrieve);
$TV_ref = lock_retrieve("tvdata.storable");
```

Модуль Storable также предлагает функцию dclone, создающую «глубокие» копии многоуровневых структур данных. Как правило, воспользоваться этой функцией намного проще, чем писать собственный вариант.

```
use Storable qw(dclone);
%TV_copy = % { dclone \%TV };
```

Другие фокусы с применением модулей Data::Dumper и Storable описаны в документации этих модулей.

Есть много других решений записи данных в разных форматах – от упакованного двоичного (очень быстрого) до XML (широко совместимого). Отличным выбором может оказаться промежуточное решение – формат YAML, обладающий вполне приемлемой удобочитаемостью. Сегодня же посетите ближайшее зеркало CPAN!

10

Пакеты

В этой главе мы добрались до приятных вещей, поскольку речь пойдет о проектировании программного обеспечения. В разговоре о качественном проектировании программного обеспечения никак не обойтись без таких базовых для проектирования тем, как Лень, Нетерпеливость и Высокомерие.

Мы все попадали в ловушку и бездумно копировали код (cut-and-paste), когда следовало бы определить абстракцию более высокого уровня, пусть даже такую простую, как обычный цикл или подпрограмма.[1] Многим наверняка доводилось впадать в противоположную крайность, нагромождая горы абстракций более высокого уровня, когда надо было лишь скопировать код через буфер обмена.[2] И все же большинству из нас следует подумать о том, чтобы использовать абстракции почаще.

Промежуточное положение занимают приверженцы взвешенного подхода к применению абстракций, которые, однако, слишком рано бросаются создавать собственные абстракции, когда следовало бы повторно использовать имеющийся код.[3] Испытав соблазн сделать что-либо из перечисленного, сделайте перерыв и подумайте, что в долгосрочной перспективе окажется лучше всего для вас и вашего ближнего. Если вы хотите излить свою творческую энергию в программе, почему бы при этом не сделать мир совершеннее? (Даже если вашей единственной целью является *успех* программы, вы должны убедиться, что она займет нужную экологическую нишу.)

Первый шаг на пути к экологически устойчивому программированию очень прост: не надо мусорить в парке. Когда вы пишете некоторый код, задумайтесь о создании для него собственного пространства имен, чтобы ваши переменные и функции никому не помешали, и наоборот. Пространство имен немного похоже

---

1 Это разновидность Ложной *Лени*.

2 Это разновидность Ложного *Высокомерия*.

3 Вы угадали: это Ложная *Нетерпеливость*. Но если вы намерены изобретать заново колесо, постарайтесь хотя бы, чтобы ваше колесо было более удачным.

на ваш собственный дом, в котором можно быть сколь угодно неряшливым, соблюдая, конечно, во взаимодействии с прочими гражданами достаточные приличия. В Perl пространство имен называется *пакетом* (*package*). Пакеты – это базовые кирпичики, из которых сооружаются понятия модулей и классов, находящиеся на более высоком уровне.

Как и понятие «дом», понятие «пакет» несколько туманно. Пакеты не зависят от файлов. Можно иметь много пакетов в одном файле и один пакет, охватывающий несколько файлов, точно так же, как ваш дом может быть маленькой мансардой в большом здании (если вы – голодающий художник), а может состоять из нескольких зданий (если вас зовут королева Елизавета). Но обычный размер дома – одно здание, а обычный размер пакета – один файл. Perl оказывает особую помощь тем, кто хочет поместить пакет в один файл, если только они согласны дать файлу то же имя, что и пакету, и использовать расширение *.pm*, что представляет собой сокращение от «perl module». *Модуль* является в Perl фундаментальной единицей повторного использования кода. На практике обращение к модулю осуществляется с помощью инструкции use, управляющей импортом подпрограмм и переменных из модулей. Все случаи применения use, встречавшиеся в тексте книги до сих пор, были примерами повторного использования модулей.

Архив Perl (Comprehensive Perl Archive Network, или CPAN) – то место, где вам следует публиковать свои модули, если вы считаете, что они могут оказаться полезными другим. Процветание Perl вызвано стремлением программистов делиться плодами своего труда с обществом. Естественно, в CPAN можно найти модули, заботливо помещенные туда другими для всеобщего пользования. Подробности читайте в главе 19 и на сайте *http://www.cpan.org*.

Тенденцией последних примерно 25 лет была разработка языков программирования, навязывающих состояние паранойи. Предполагается, что каждый модуль вы должны программировать так, будто он находится в положении осады. Конечно, существуют феодальные культуры, где такой подход уместен, но не все культуры таковы. В культуре Perl, к примеру, предполагается, что вы не залезаете в чужой дом потому, что вас не приглашали, а не потому, что на окнах решетки.[1]

Эта книга не посвящена объектно-ориентированным технологиям, и мы не собираемся обращать вас в неистового фанатика объектной ориентированности, даже если вы этого хотите. Для этого уже написано множество книг. Философия Perl в отношении объектно-ориентированной разработки соответствует его философии в отношении всего прочего: прибегать к объектно-ориентированной разработке там, где это имеет смысл, и избегать там, где этого смысла нет. Выбор за вами.

На жаргоне ООП каждый объект принадлежит объединению, называемому *классом*. В Perl классы, пакеты и модули так тесно связаны, что новичкам часто кажется, будто они взаимозаменяемы. Типичный класс реализуется модулем, в котором определен пакет, имя которого совпадает с именем класса. Все это мы разъясним в нескольких последующих главах.

Применение модулей дает преимущество непосредственного повторного использования программного обеспечения. А классы обеспечивают преимущество косвенного повторного использования программного обеспечения, когда один класс

---

[1] Однако в Perl есть и решетки, если они вам нужны. См. раздел «Работа с небезопасным кодом» главы 20.

использует другой посредством наследования. А еще классы позволяют осуществлять чистое взаимодействие между пространствами имен. Доступ ко всему, что находится в классе, осуществляется косвенным образом, изолируя класс от внешнего мира.

Как упоминалось в главе 8, объектно-ориентированное программирование в Perl реализуется через ссылки, объекты которых «знают», к какому классу они принадлежат. Фактически, теперь, когда вы знаете о ссылках, вы знаете почти обо всем сложном, что связано с объектами. В остальном «пальцы сами ложатся на нужные ноты», как сказал бы пианист. Конечно, нужно немного попрактиковаться.

Одно из основных упражнений для пальцев состоит в том, чтобы научиться защищать различные фрагменты кода от непреднамеренного взаимного воздействия на переменные. Каждый участок кода относится к определенному *пакету*, который определяет, какие переменные и подпрограммы ему доступны. Когда Perl встречает фрагмент кода, то компилирует его в область, которая называется *текущим пакетом*. Первоначальный текущий пакет называется «main», но в любой момент вы можете переключить текущий пакет на другой с помощью объявления package. Текущий пакет определяет, какая таблица имен используется для поиска переменных, подпрограмм, дескрипторов ввода/вывода и форматов.

Таблицы имен

Содержимое пакета в совокупности называется *таблицей имен* (*symbol table*). Таблица имен хранится в хеше, имя которого совпадает с именем пакета, но дополнено двумя двоеточиями. Таким образом, именем таблицы для main служит %main::. Поскольку main является пакетом по умолчанию, Perl допускает сокращенное имя %:: для %main::.

Аналогично, таблица имен для пакета Red::Blue носит имя %Red::Blue::. Таблица имен main содержит все остальные таблицы имен верхнего уровня, включая саму себя, поэтому %Red::Blue:: одновременно является %main::Red::Blue::.

Когда мы говорим, что таблица имен содержит другую таблицу имен, мы имеем в виду, что она содержит ссылку на другую таблицу имен. Поскольку main является пакетом верхнего уровня, он содержит ссылку на самого себя, поэтому %main:: это то же самое, что и %main::main::, и %main::main::main::, и так до бесконечности. Этот особый случай важно учитывать при написании кода, который обходит все таблицы имен.

В хеше таблицы имен каждая пара ключ/значение устанавливает соответствие между именем переменной и ее значением. Ключи представляют собой символьные идентификаторы, а значения являются соответствующими typeglob. Поэтому посредством обозначения **NAME* typeglob мы, в действительности, просто обращаемся к значению в хеше, который содержит таблицу имен текущего пакета. На практике следующие строки имеют (почти) одинаковый эффект:

```
*sym = *main::variable;
*sym = $main::{"variable"};
```

Первая строка более эффективна, потому что доступ к таблице имен main может осуществляться уже на этапе компиляции. Она также создает новый typeglob

с указанным именем, если его еще не существует, тогда как вторая форма этого не делает.

Поскольку пакет представляет собой хеш, можно найти ключи пакета и получить все его переменные. А так как значения хеша являются typeglob, разыменовывать их можно несколькими способами. Например, так:

```
foreach $symname (sort keys %main::) {
    local *sym = $main::{$symname};
    print "\$$symname определен\n" if defined $sym;
    print "\@$symname не нуль\n"   if         @sym;
    print "\%$symname не нуль\n"   if         %sym;
}
```

Все пакеты доступны (прямо или косвенно) через пакет main, поэтому можно написать код Perl, который обойдет все переменные программы. Именно это делает отладчик Perl, когда вы запрашиваете у него снимок переменных посредством команды V. Заметьте, что при этом не будут видны переменные, объявленные с помощью my, поскольку они не зависят от пакетов, но будут видны переменные, объявленные через our. См. главу 18.

Ранее мы сказали, что только идентификаторы могут храниться в пакетах, отличных от main. Это маленький обман: в качестве ключа в хеше таблицы имен можно указать любую строку, просто Perl не допустит непосредственного использования строки, не являющейся идентификатором:

```
$!@#$%          = 0;          # НЕВЕРНО, синтаксическая ошибка.
${'!@#$%'}      = 1;          # порядок, хотя имя не полное.

${'main::!@#$%'} = 2;         # Можно уточнять имя в строке.
print ${ $main::{'!@#$%'} }   # порядок. выводит 2!
```

Присваивание typeglob осуществляет операцию создания псевдонима; т.е. после выполнения

```
*dick = *richard;
```

переменные, подпрограммы, форматы, дескрипторы файлов и каталогов, доступные через идентификатор richard, оказываются доступными и через имя dick. Чтобы создать псевдоним только для конкретной переменной или подпрограммы, используйте ссылку:

```
*dick = \$richard;
```

В результате $richard и $dick становятся одной и той же переменной, но @richard и @dick остаются разными массивами. Хитро, а?

Вот как работает Exporter при импорте имен из одного пакета в другой. Например,

```
*SomePack::dick = \&OtherPack::richard;
```

импортирует функцию &richard из пакета OtherPack в пакет SomePack, делая ее доступной под именем &dick. (Модуль Exporter описан в следующей главе.) Если предварить это присваивание ключевым словом local, псевдоним будет действовать только до конца текущей динамической области видимости.

Этот механизм может применяться для извлечения ссылки из подпрограммы, делая объект ссылки доступным в качестве соответствующего типа данных:

```
*units = populate()            # Присвоить \%newhash объекту typeglob
print $units{kg};              # Выведет 70; разыменование не требуется!

sub populate {
    my %newhash = (km => 10, kg => 70);
    return \%newhash;
}
```

Аналогично можно передать ссылку в подпрограмму и использовать ее без разыменования:

```
%units = (miles => 6, stones => 11);
fillerup( \%units );         # Передача ссылки
print $units{quarts};        # Выведет 4

sub fillerup {
    local *hashsym = shift;  # Присвоить \%units объекту typeglob
    $hashsym{quarts} = 4;    # Действует на %units; разыменование не требуется!
}
```

Это хитрые приемы для передачи ссылок без лишних затрат, когда вы не хотите явно их разыменовывать. Обратите внимание на то, что оба приема работают только с переменными пакета; их нельзя использовать, если объявить `%units` как `my`.

С помощью таблиц имен можно также создавать «константы»-скаляры:

```
*PI = \3.14159265358979;
```

Теперь изменить `$PI` нельзя, что, вообще-то, правильно. Это не то же самое, что подпрограмма-константа, оптимизируемая на этапе компиляции. Подпрограмма-константа – это такая подпрограмма, в прототипе которой указано, что она не принимает аргументов и возвращает постоянное выражение; подробности можно найти в разделе «Подставляемые функции-константы» главы 7. Удобным сокращением является использование прагмы `use constant` (см. главу 29):

```
use constant PI => 3.14159;
```

Технически при этом используется место для подпрограммы `*PI`, а не место для скаляра, как ранее. Это эквивалентно более компактной (но не столь прозрачной) записи:

```
*PI = sub () { 3.14159 }
```

В любом случае, полезно знать эту идиому: присваивание `sub {}` объекту typeglob является способом дать имя анонимной подпрограмме на этапе выполнения.

Присвоить ссылку на typeglob другой переменной typeglob (`*sym = \*oldvar`) означает то же самое, что присвоить typeglob целиком, потому что Perl автоматически разыменовывает ссылку на typeglob. А когда объекту typeglob присваивается обычная строка, она становится именем typeglob в целом, потому что Perl ищет строку в текущей таблице имен. Все нижеследующие строки эквивалентны между собой, но первые две вычисляют элемент таблицы имен на этапе компиляции, а две последние – на этапе выполнения:

```
*sym = *oldvar;
*sym = \*oldvar;       # автоматическое разыменование
```

```
*sym = *{"oldvar"};     # явный поиск в таблице имен
*sym = "oldvar";        # неявный поиск в таблице имен
```

Осуществляя любое из следующих присваиваний, мы заменяем только одну ссылку внутри typeglob:

```
*sym = \$frodo;
*sym = \@sam;
*sym = \%merry;
*sym = \&pippin;
```

Если взглянуть на это со стороны, то typeglob можно рассматривать как хеш, содержащий записи для различных типов переменных. В данном случае ключи фиксированы, поскольку typeglob может содержать ровно один скаляр, один массив, один хеш и т.д. Но можно извлекать отдельные ссылки:

```
*pkg::sym{SCALAR}  # то же, что \$pkg::sym
*pkg::sym{ARRAY}   # то же, что \@pkg::sym
*pkg::sym{HASH}    # то же, что \%pkg::sym
*pkg::sym{CODE}    # то же, что \&pkg::sym
*pkg::sym{GLOB}    # то же, что \*pkg::sym
*pkg::sym{IO}      # внутренний дескриптор файла/каталога,
                   # нет прямого эквивалента
*pkg::sym{NAME}    # "sym" (не ссылка)
*pkg::sym{PACKAGE} # "pkg" (не ссылка)
```

Выражения *foo{*PACKAGE*} и *foo{*NAME*} позволяют узнать, каковы имя и пакет для элемента таблицы имен *foo. Это может оказаться полезным в подпрограмме, получающей typeglob в качестве аргументов:

```
sub identify_typeglob {
    my $glob = shift;
    print "Вы передали мне ", *{$glob}{PACKAGE}, '::', *{$glob}{NAME}, "\n";
}
identify_typeglob(*foo);
identify_typeglob(*bar::glarch);
```

В результате будет выведено:

```
Вы передали мне main::foo
Вы передали мне bar::glarch
```

Форму записи *foo{*THING*} можно использовать для получения ссылок на отдельные элементы *foo. Подробности см. в разделе «Ссылки на таблицы имен» главы 8.

Этот синтаксис служит, в основном, для получения внутренней ссылки на дескриптор файла или каталога, поскольку другие внутренние ссылки можно получить иным способом. (Прежнее *foo{*FILEHANDLE*} более не поддерживается в значении *foo{*IO*}.) Но мы решили, что следует обобщить его, поскольку он выглядит довольно симпатично. Скорее всего, вам не нужно все это запоминать, если только вы не собираетесь написать новый отладчик Perl.

Квалифицированные имена

На *идентификаторы*[1] в других пакетах можно ссылаться, предваряя идентификаторы именем пакета и парой двоеточий: `$Пакет::Переменная`. Это мы называем *«квалифицировать»* идентификатор. Если имя пакета опущено, предполагается, что речь идет о пакете `main`. Это значит, что `$::sail` эквивалентно `$main::sail`.[2]

В прошлом разделителем идентификатора и пакета служила одинарная кавычка, поэтому в старых программах на Perl можно увидеть переменные типа `$main'sail` и `$somepack'horse`. Но сейчас предпочтительным разделителем является двойное двоеточие – отчасти потому, что его легче читать человеку, отчасти потому, что его легче читать макросам *emacs*. Он также создает у программистов на C++ впечатление, будто они понимают, что происходит, тогда как одинарная кавычка решала ту же задачу для программистов на языке Ada. Поскольку старомодный синтаксис все еще поддерживается для обратной совместимости, в строке вроде `"This is $owner's house"` будет выполнено обращение к `$owner::s`; т.е. переменной `$s` из пакета `owner`, чего вы, возможно, не желали. Двусмысленности можно избежать при помощи фигурных скобок, например `"This is ${owner}'s house"`.

Двойное двоеточие может применяться для связывания в цепочку идентификаторов в имени пакета: `$Red::Blue::var`. Это обозначает переменную `$var`, принадлежащую пакету `Red::Blue`. Пакет `Red::Blue` не имеет никакого отношения к возможно существующим пакетам `Red` или `Blue`. То есть, отношения между `Red::Blue` и `Red` или `Blue` могут существовать для того, кто пишет или использует программу, но они не существуют с точки зрения Perl. (Кроме того обстоятельства, что в текущей реализации таблица имен `Red::Blue` хранится в таблице имен `Red`. Но язык Perl этим обстоятельством напрямую не пользуется.)

Когда-то переменные, начинавшиеся символом подчеркивания, принудительно помещались в пакет `main`, но потом мы решили, что авторам пакетов удобнее при помощи ведущего символа подчеркивания обозначать полузакрытые идентификаторы, предназначенные только для использования внутри пакета. (Подлинно закрытые переменные могут быть объявлены как лексические с областью видимости «файл», но это лучше всего работает, когда между пакетом и модулем имеется взаимно однозначное соответствие, что случается часто, но не является обязательным требованием.)

Хеш `%SIG` (предназначенный для перехвата сигналов; см. главу 15) тоже является особым. Если определить обработчик сигнала как строку, то она будет указывать на подпрограмму в пакете `main`, если только явно не указано имя другого пакета. Указывайте полностью квалифицированное имя обработчика сигнала, если

---

1 Под идентификаторами мы подразумеваем имена, служащие ключами таблицы имен для доступа к скалярным переменным, переменным массивов, переменным хешей, подпрограммам, дескрипторам файлов и каталогов и форматам. Если говорить о синтаксисе, метки тоже являются идентификаторами, но они не помещаются в какую-либо таблицу имен, а непосредственно прикрепляются к командам программы. Метки не могут квалифицироваться именем пакета.

2 Чтобы избежать возможной путаницы, в имени переменной типа `$main::sail` мы используем термин «идентификатор» для `main` и `sail`, но не для `main::sail`. Мы называем такую конструкцию именем переменной, потому что идентификаторы не могут содержать двоеточий.

хотите указать конкретный пакет, или вообще избегайте использования строк, осуществляя вместо этого присваивание typeglob или ссылки на функцию:

```
$SIG{QUIT} = "Pkg::quit_catcher"  # квалифицированное имя обработчика
$SIG{QUIT} = "quit_catcher";      # подразумевается "main::quit_catcher"
$SIG{QUIT} = *quit_catcher;       # устанавливает sub из текущего пакета
$SIG{QUIT} = \&quit_catcher;      # устанавливает sub из текущего пакета
$SIG{QUIT} = sub { print "Перехвачен SIGQUIT\n" }; # анонимная sub
```

Пакет по умолчанию

Пакетом по умолчанию является main, то же имя, что назначено главной функции программы на языке C. Если явно не определено иное, все переменные оказываются в этом пакете. Следующие строки являются идентичными:

```
#!/usr/bin/perl

$name       = 'Amelia';
$main::name = 'Amelia';

$type       = 'Camel';
$main::type = 'Camel';
```

В области действия прагмы strict необходимо явно определять пакет, поскольку эта прагма не позволяет использовать необъявленные переменные:

```
#!/usr/bin/perl

use v5.12;

$name       = 'Amelia';    # ошибка компиляции
$main::name = 'Amelia';

$type       = 'Camel';     # ошибка компиляции
$main::type = 'Camel';
```

В таблице имен пакета хранятся только идентификаторы (имена, начинающиеся с буквы или символа подчеркивания). Все остальные имена хранятся в пакете main, в том числе все небуквенные переменные, такие как $!, $? и $_.[1] Кроме того, идентификаторы STDIN, STDOUT, STDERR, ARGV, ARGVOUT, ENV, INC и SIG, если они не квалифицированы, принудительно помещаются в пакет main, даже если используются для других целей, нежели их встроенные тезки. Не называйте свои пакеты m, s, y, tr, q, qq, qr, qw или qx, если только не хотите нажить себе множество неприятностей. Например, вы не сможете использовать квалифицированную форму идентификатора в качестве дескриптора файла, поскольку она будет интерпретирована как поиск по шаблону, подстановка или транслитерация.

---

[1] Однако, начиная с v5.10 допускается создание переменных $_ с лексической областью видимости.

Изменение пакета

Понятие «текущего пакета» относится и к этапу компиляции, и к этапу выполнения. Большинство операций поиска имен переменных происходит на этапе компиляции, а поиск на этапе выполнения случается при разыменовании символических переменных, а также когда новые фрагменты кода анализируются при выполнении eval. В частности, когда eval применяется к строке, Perl знает, в каком пакете была вызвана eval, и распространяет действие этого пакета на контекст обрабатываемой строки. (Конечно, внутри строки eval всегда можно переключиться в другой пакет, поскольку строка eval считается блоком, как и файл, загруженный с помощью do, require или use.)

По этой причине каждое объявление package должно содержать полное имя пакета. Для имен пакетов не существует каких-либо неявных «префиксов», даже если пакет объявляется внутри объявления некоторого другого пакета.

С другой стороны, eval может узнать, в каком пакете находится ее вызов, с помощью специального символа __PACKAGE__, в котором содержится имя текущего пакета. Поскольку с этим символом можно обращаться как со строкой, он может фигурировать в символической ссылке для доступа к переменной пакета. Но в таком случае, вероятно, следовало бы объявить переменную с ключевым словом our и обращаться к ней как к лексической переменной.

Любые переменные, не объявленные с ключевым словом my, ассоциируются с пакетом – даже кажущиеся вездесущими переменные типа $_ и %SIG. Другие переменные ассоциируются с текущим пакетом, если они не квалифицированы:

```
$name = 'Amelia';           # имя в текущем пакете

$Animal::name = 'Camelia'; # имя в пакете Animal
```

Объявление package изменяет пакет по умолчанию до конца области видимости (блока, файла или eval) или пока не встретится другое объявление package на том же уровне, замещающее прежний пакет (что является распространенной практикой):

```
package Animal;

$name = 'Camelia'           # $Animal::name
```

Важно отметить и еще раз подчеркнуть, что объявление package не создает область видимости, поэтому оно не способно скрыть лексические переменные в той же области видимости:

```
my $type = 'Camel';

package Animal;

print "Type is $type\n";   # лексическая $type, поэтому "Camel"
$type = 'Ram';

package Zoo;

print "Type is $type\n";   # лексическая $type, поэтому "Ram"
```

Чтобы отдать предпочтение переменной с тем же именем, принадлежащей пакету, используйте our. Но будьте осторожны. В результате переменная из текущего пакета будет использоваться по умолчанию до конца области видимости, даже если пакет по умолчанию изменится:

```
my $type = 'Camel';

package Animal;

our $type = 'Ram';
print "Type is $type\n";   # переменная $type из пакета, поэтому "Ram"

package Zoo;

print "Type is $type\n";   # переменная $type из пакета Animal, поэтому "Ram
```

В пакете Zoo имя $type все еще представляет переменную $Animal::type. Область действия объявления our простирается до конца области видимости, а не до конца области действия объявления пакета. Это может вносить некоторую путаницу. Запомните, что объявление package изменяет только имя пакета по умолчанию: оно не начинает и не заканчивает область видимости. После изменения пакета все последующие необъявленные идентификаторы будут помещаться в таблицу символов, принадлежащую текущему пакету. Вот так-то.

Обычно объявление package является первой командой в файле, который должен включаться посредством require или use. Но, опять-таки, это лишь соглашение. Объявление package можно поместить в любое место, где допустим оператор. Можно даже поместить его в конце блока, и тогда оно вообще не будет иметь никакого эффекта. Можно переключать пакет в нескольких местах; объявление пакета просто выбирает таблицу имен, которая будет использоваться компилятором в оставшейся части блока. Этот способ позволяет создать пакет, охватывающий более одного файла.

В последних версиях Perl можно указывать версию пакета в строке с объявлением package:

```
package Zoo v3.1.4;
```

Кроме того, в Perl v5.14 и более поздних версий допускается заключать пакеты в фигурные скобки, что делает их более похожими на блоки. Это позволяет ограничить область видимости пакета границами блока. Используя эту особенность, можно избежать проблемы утечки имен, описанной выше:

```
my $type = 'Camel';

package Animal {
    our $type = 'Ram';
    print "Type is $type\n";   # переменная $type из пакета, поэтому "Ram"
}

package Zoo v3.1.4 {
    print "Type is $type\n";   # внешняя переменная $type, поэтому "Camel"
}
```

Автозагрузка

Обычно невозможно вызвать подпрограмму, которая не определена. Однако если в пакете с неопределенной подпрограммой (или, если речь идет о методе объекта, в пакете любого из базовых классов объекта) содержится подпрограмма с именем AUTOLOAD, подпрограмма AUTOLOAD вызывается с теми же аргументами, которые были бы переданы исходной подпрограмме. Можно определить подпрограмму AUTOLOAD так, чтобы она возвращала значения как обычная подпрограмма или заставить ее определить отсутствующую подпрограмму, а затем вызвать новую подпрограмму, как будто она существовала до вызова AUTOLOAD.

Полностью квалифицированное имя исходно не существовавшей подпрограммы чудесным образом появляется в $AUTOLOAD – глобальной переменной того же пакета, в котором находится подпрограмма AUTOLOAD. Вот простой пример, который деликатно предупреждает вас о вызове неопределенной подпрограммы, вместо того чтобы осуществить выход:

```
sub AUTOLOAD {
    our $AUTOLOAD;
    warn "Попытка вызвать $AUTOLOAD не удалась.\n";
}

blarg(10);      # наша $AUTOLOAD получит значение main::blarg
print "Продолжаем работу!\n";
```

Либо можно вернуть значение от имени неопределенной подпрограммы:

```
sub AUTOLOAD {
    our $AUTOLOAD;
    return "Я вижу $AUTOLOAD(@_)\n";
}

print blarg(20);    # выведет: Я вижу main::blarg(20)
```

Ваша подпрограмма AUTOLOAD может загрузить определение для неопределенной подпрограммы с помощью eval или require либо прибегнуть к обсуждавшемуся выше приему с присвоением glob, а затем выполнить эту неопределенную подпрограмму с применением специальной формы goto, которая может без следа стереть запись активации подпрограммы AUTOLOAD. Сейчас мы определим подпрограмму, присвоив замыкание переменной glob:

```
sub AUTOLOAD {
    my $name = our $AUTOLOAD;
    *$AUTOLOAD = sub { print "Я вижу $name(@_)\n" };
    goto &$AUTOLOAD; # Запустить новую подпрограмму
}

blarg(30);          # выведет: Я вижу main::blarg(30)
glarb(40);          # выведет: Я вижу main::glarb(40)
blarg(50);          # выведет: Я вижу main::blarg(50)
```

Стандартный модуль AutoSplit применяется разработчиками модулей для автоматического разделения своих модулей на отдельные файлы (имена которых заканчиваются *.al*), в каждом из которых содержится одна подпрограмма. Эти файлы помещаются в каталог *auto/* в системной библиотеке Perl, после чего их можно

автоматически загружать по требованию с помощью стандартного модуля AutoLoader.

Аналогичный подход применяется в модуле SelfLoader, за исключением того, что он автоматически загружает функции из собственной области DATA файла, что менее эффективно в одних отношениях и более эффективно в других. Автозагрузка функций Perl с помощью AutoLoader и SelfLoader аналогична динамической загрузке скомпилированных функций C посредством модуля DynaLoader, за исключением того, что минимальная единица автозагрузки – функция, а минимальная единица динамической загрузки – целый модуль, и при этом обычно компонуется сразу много функций C или C++. (Заметим, что многие программисты Perl прекрасно обходятся без модулей AutoSplit, AutoLoader, SelfLoader или DynaLoader. Просто знайте, что такие модули существуют, на случай, если не сможете прекрасно обходиться без них.)

Можно неплохо развлечься, работая с подпрограммами AUTOLOAD, служащими обертками для других интерфейсов. Давайте, например, потребуем, чтобы для каждой неопределенной функции вызывалась команда system с теми же аргументами. Нужно сделать лишь следующее:

```
sub AUTOLOAD {
    my $program = our $AUTOLOAD;
    $program =~ s/.*:://;       # обрезать имя пакета
    system($program, @_);
}
```

(Поздравляем, вы реализовали элементарную форму модуля Shell, который входит в стандартную поставку Perl.) Вызвать свой автозагрузчик (в ОС UNIX) можно так:

```
date();
who('am', 'i');
ls('-l');
echo("Abadugabudabuda...");
```

На самом деле, если предварительно объявить функции, которые требуется вызывать таким способом, можно действовать так, как если бы они были встроенными, и опускать скобки при вызове:

```
sub date (;$$);        # Допускает от нуля до двух аргументов.
sub who (;$$$$);       # Допускает от нуля до четырех аргументов
sub ls;                # Допускает любое число аргументов
sub echo ($@);         # Допускает не менее одного аргумента

date;
who "am", "i";
ls "-l";
echo "That's all, folks!";
```

Начиная с версии v5.8, AUTOLOAD может иметь атрибут :lvalue.

```
package Chameau;
use v5.14;

sub new { bless {}, $_[0] }
```

```
sub AUTOLOAD :lvalue {
    our $AUTOLOAD;
    my $method = $AUTOLOAD =~ s/.*:://r;
    $_[0]->{$method};
}

1;
```

Этот метод можно использовать для организации доступа или присваивания ему:

```
use v5.14;
use Chameau;

my $chameau = Chameau->new;

$chameau->awake = 'yes';

say $chameau->awake;
```

Или сделать последнее значение символической ссылкой:

```
package Trampeltier;

sub new { bless {}, $_[0] }
sub AUTOLOAD :lvalue { no strict 'refs'; *{$AUTOLOAD} }

1;
```

чтобы получить возможность определять метод через присваивание:

```
use Trampeltier;

my $trampeltier = Trampeltier->new;

$trampeltier->name = sub { 'Amelia' }
```

Однако, мы не уверены, что вы загоритесь желанием поступать так.

11

Модули

В языке Perl модуль является фундаментальной единицей повторного использования кода. С точки зрения реализации, модуль – это просто пакет, определенный в одноименном файле с расширением *.pm*. В этой главе мы узнаем, как пользоваться сторонними модулями и создавать собственные.

В поставку Perl входят сотни модулей, которые можно найти в каталоге *lib* дистрибутива Perl, или в каталоге, выбранном на этапе сборки *perl*. Получить список каталогов с модулями позволяет ключ *-V*:

```
% perl -V
Summary of my perl5 (revision 5 version 14 subversion 1) configuration:

...
Built under darwin
Compiled at Jul 5 2011 21:43:59
@INC:
  /usr/local/perl/lib/site_perl/5.14.2/darwin-2level
  /usr/local/perl/lib/site_perl/5.14.2
  /usr/local/perl/lib/5.14.2/darwin-2level
  /usr/local/perl/lib/5.14.2
  .
```

Получить полный список модулей, поставляемых вместе с *perl*, можно с помощью команды *corelist*, которая также входит в состав дистрибутива Perl:

```
% corelist -v 5.014
```

Кроме того, для всех стандартных модулей существует обширная электронная документация, которая (о, ужас!), скорее всего, окажется более актуальной, чем данная книга. Попробуйте применить команду *perldoc* для чтения документации:

```
% perldoc Digest::MD5
```

Архив Perl «Comprehensive Perl Archive Network» (CPAN) являет собой всемирное хранилище модулей, созданных сообществом Perl, и обсуждается в главе 19. См. также *http://www.cpan.org*.

Загрузка модулей

Модули бывают двух видов: традиционные и объектно-ориентированные. Традиционные модули определяют подпрограммы и переменные, которые вызывающая сторона может импортировать и, конечно, использовать. Объектно-ориентированные модули действуют как определения классов, и доступ к ним осуществляется через вызовы методов, как описывается в главе 12. Некоторые модули совмещают эти подходы.

Модули Perl обычно включаются в программу командой:

```
use MODULE;
```

что эквивалентно такой конструкции:

```
BEGIN {
    require MODULE;
    MODULE->import();
}
```

Загрузка выполняется на этапе компиляции, поэтому весь программный код в модулях выполняется в процессе компиляции. Обычно это не является проблемой, потому что большая часть кода в модулях заключена в подпрограммы и методы. Некоторые модули могут загружать дополнительные модули, XS-код[1] и другие компоненты. Поскольку директива `use` **выполняется сразу, как только попалась на глаза Perl, все изменения в** `@INC` **должны предшествовать директиве** `use`**. Возможно, вам требуется прагма** `lib` **(см. главу 29), которая также загружается с помощью** `use`**.**

Если возникнет необходимость загрузить модуль на этапе выполнения, например, чтобы отложить его включение до выполнения некоторой подпрограммы, нуждающейся в нем, можно воспользоваться функцией `require`**:**

```
require MODULE;
```

Аргумент `MODULE` **должен быть именем пакета, которое будет преобразовано в имя файла модуля. Директива** `use` **преобразует символы** `::` **в** `/` **и добавляет в конец** ***.pm*****. Она пытается отыскать полученное имя в** `@INC`**. Если предположить, что модуль называется** `Animal::Mammal::HoneyBadger`**, директива** `use` **будет искать файл** ***Animal/Mammal/HoneyBadger.pm*****. После загрузки модуля путь к соответствующему файлу будет добавлен в** `%INC`**. Файл загружается только один раз. Перед тем как загрузить файл, Perl проверит содержимое** `%INC`**, чтобы убедиться, что файл не был загружен ранее. Если выяснится, что файл был загружен, Perl использует загруженную версию.**

Файл можно загрузить напрямую, посредством `require`**, указав корректный путь (который может зависеть от платформы):**

```
require FILE;
require 'Animal/Mammal/HoneyBadger.pm';
```

---

[1] XS-код (eXternal Subroutines – внешние подпрограммы) – это модули, написанные на компилируемых языках программирования, обычно C, и пригодные для импортирования из программ и модулей на языке Perl. – *Прим. перев.*

В целом, однако, предпочтительнее использовать use, а не require, потому что в этом случае поиск модулей происходит во время компиляции и, если имеются ошибки, о них можно будет узнать раньше.

Некоторые модули предлагают дополнительную функциональность в виде списка импорта. Этот список превращается в список аргументов для функции import:

```
use MODULE LIST;
```

эквивалентно:

```
BEGIN {
    require MODULE;
    MODULE->import(LIST);
}
```

Метод import модуля может выполнять любые действия, но в большинстве модулей он используется для определения номера версии, унаследованного от Exporter, о котором мы поговорим ниже. Обычно метод import добавляет символы (подпрограммы и переменные) в текущее пространство имен, чтобы обеспечить их доступность в оставшейся части единицы компиляции. Некоторые модули имеют список импорта по умолчанию.

Например, модуль Hash::Util экспортирует несколько символов для выполнения специальных операций с хешами. Следующая директива use добавит символ lock_keys, который окажется доступен в оставшейся части единицы компиляции:

```
use Hash::Util qw(lock_keys);

lock_keys( my %hash, qw(name location) );
```

Даже при вызове без списка *LIST* директива use может импортировать символы, включенные в список импорта по умолчанию.[1] При загрузке модуля File::Basename автоматически импортируются функции basename, dirname и fileparse:

```
use File::Basename;

say basename($ARGV[0]);
```

Если ничего не требуется импортировать из модуля, можно явно указать пустой список:

```
use MODULE ();
```

Иногда бывает желательно использовать конкретную версию модуля (или версию, не ниже определенной), чтобы избежать известных проблем, имеющихся в прежних версиях, или использовать новейший API, не совместимый со старыми версиями:

```
use MODULE VERSION LIST;
```

Обычно вполне подходит версия равная или выше *VERSION*. Нельзя указать точную версию или диапазон версий. Однако модуль может принять иное решение, поскольку в действительности VERSION – это метод.

---

[1] В наши дни это считается несколько невежливым. Вынуждая программиста явно указывать, что он хочет импортировать, вы поможете ему избежать конфликтов между двумя разными модулями, экспортирующими одинаковые имена.

Выгрузка модулей

Директива no противоположна директиве use. Вместо метода import она вызывает unimport. Последний может делать все, что угодно. Синтаксис выгрузки модуля точно такой же:

```
no MODULE;
no MODULE LIST;
no MODULE VERSION;
no MODULE VERSION LIST;
```

Иногда бывает желательно, чтобы некоторые символы были доступны лишь непродолжительное время. Например, при загрузке модуля Moose, объектной системы, основанной на встроенных возможностях Perl, импортируется множество вспомогательных методов. Метод has объявляет атрибуты, но как только эти имена станут не нужны, их можно удалить. В конце раздела, где они используются, эти имена можно выгрузить директивой no:

```
package Person;
use Moose;

has "first_name" => (is => "rw", isa => "Str");
has "last_name"  => (is => "rw", isa => "Str");

sub full_name {
    my $self = shift;
    $self->first_name . " " . $self->last_name
}
no Moose; # удалит ключи из пакета Person
```

Чтобы временно отключить прагму strict, ее можно выгрузить, как показано ниже. Используйте этот прием с как можно меньшими областями видимости, чтобы не пропустить другие проблемы:

```
my $value = do {
    no strict "refs";

    ${ "${class}::name" } # символическая ссылка
};
```

Аналогично может потребоваться временно отключить вывод некоторых предупреждений, чего можно добиться, выгрузив их:

```
use warnings;
{
    no warnings 'redefine';
    local *badger = sub { ... };
    ...;
}
```

Создание модулей

В этой главе мы просто покажем фрагмент модуля. Все, что касается создания *дистрибутивов*, будет описано в главе 19.

Именование модулей

Выбор хорошего имени для модуля является важным этапом в создании модуля. Как только имя будет выбрано и люди начнут использовать ваш модуль, вам придется жить с этим именем практически вечно, так как пользователи обычно не стремятся обновлять свой код. Если вы собираетесь опубликовать свой модуль в архиве CPAN, желательно, чтобы другие легко могли отыскать его. Некоторые правила именования модулей можно найти на странице PAUSE (*http://pause.perl.org/pause/query?ACTION=pause_namingmodules*).

Имя модуля должно начинаться заглавной буквой, если только оно не используется в качестве прагмы. Модули прагм (глава 29) действуют подобно директивам компилятора (дают советы компилятору), поэтому мы резервируем нижний регистр для имен директив (директивных модулей) для использования в дальнейшем.

Если вы желаете сделать модуль закрытым модулем, имя которого никогда не должно конфликтовать с именами модулей в стандартной библиотеке или в CPAN, можно воспользоваться пространством имен `Local`. Это пространство имен не запрещено в архиве CPAN, но по общепринятому соглашению там не используется.

Пример модуля

Выше говорилось, что модули бывают двух видов: традиционные и объектно-ориентированные. Ниже мы покажем короткие примеры модулей того и другого типа.

Проще всего продемонстрировать объектно-ориентированные модули, поскольку для их взаимодействия с пользователем требуется минимум инфраструктуры. Все операции выполняются посредством методов:

```
package Bestiary::OO 1.001;

sub new {
    my( $class, @args ) = @_;
    bless {}, $class;
}

sub camel { "Одногорбый верблюд" }

sub weight { 1024 }

### другие методы

1;
```

Программа, использующая данный модуль, будет выполнять все операции посредством методов:

```
use v5.10;
use Bestiary::OO;

my $bestiary = Bestiary::OO->new; # метод класса

say "Животное: ". $bestiary->camel(),
    " весит "     $bestiary->weight();
```

Чтобы сконструировать традиционный модуль с именем `Bestiary`, создайте файл *Bestiary.pm* со следующим содержимым:

```
package Bestiary;
use parent qw(Exporter);

our @EXPORT    = qw(camel);     # Имена, экспортируемые по умолчанию
our @EXPORT_OK = qw($weight);   # Имена, экспортируемые по запросу

### Здесь включаются ваши функции и переменные

sub camel { print "Одногорбый верблюд" }

$weight = 1024;

1; # модуль завершается выражением, возвращающим истинное значение
```

Теперь программа может сказать `use Bestiary`, чтобы получить доступ к функции `camel` (но не к переменной `$weight`), и сказать `use Bestiary qw(camel $weight)`, чтобы получить доступ как к функции, так и к переменной:

```
use v5.10;

use Bestiary qw(camel $weight);

say "Животное: ", camel(), " весит $weight";
```

Можно также создавать модули, динамически загружающие код, написанный на C, однако здесь мы не станем об этом рассказывать.

Закрытость модуля и Exporter

Perl не осуществляет автоматическое патрулирование границ между закрытыми и открытыми элементами в своих модулях – в отличие от таких языков, как C++, Java и Ada, Perl не озабочен принудительной безопасностью. Модуль Perl предпочтет, чтобы вы не входили в его комнату, потому что вас не приглашали, а не потому, что у него есть дробовик.

Между модулем и его пользователем заключается соглашение, включающее положения прецедентного права, а также письменный контракт. Прецедентное право, к примеру, устанавливает, что модуль будет воздерживаться от изменения любого пространства имен, если его об этом не просили. Письменный контракт для модуля (т.е. документация) может ставить дополнительные условия. И тогда после прочтения контракта предполагается, что, говоря `use RedefineTheWorld`, вы знаете, что изменяете мир, и готовы к возможным последствиям. Самый распространенный способ переопределения миров – использование модуля `Exporter`. Как мы увидим далее в этой главе, `Exporter` позволяет переопределять даже встроенные функции.

Если модуль загружен посредством `use`, обычно он делает некоторые переменные и функции доступными вашей программе или, точнее, ее текущему пакету. Это действие по экспортированию имен модуля (импортированию их в вашу программу) иногда называют *загрязнением* (*polluting*) вашего пространства имен. Большинство модулей использует для этого `Exporter`, поэтому в начале большинства модулей есть подобные строки:

```
use parent qw(Exporter);

require Exporter;
our @ISA = ("Exporter");
```

Эти две строки заставляют модуль наследовать класс Exporter. Наследование описывается в следующей главе, а пока достаточно знать, что наш модуль Bestiary может экспортировать имена в другие пакеты с помощью таких строк:

```
our @EXPORT      = qw($camel %wolf ram);    # Экспорт по умолчанию
our @EXPORT_OK   = qw(leopard @llama $emu); # Экспорт по запросу
our %EXPORT_TAGS = (                        # Экспорт группы
                     camelids => [qw($camel @llama)],
                     critters => [qw(ram $camel %wolf)],
                   );
```

С точки зрения экспортирующего модуля, массив @EXPORT содержит имена переменных и функций, экспортируемых по умолчанию: то, что получит ваша программа, сказав use Bestiary. Переменные и функции в @EXPORT_OK экспортируются, только когда программа специально запрашивает их посредством use. Наконец, пары ключ/значение в %EXPORT_TAGS позволяют программе включать отдельные группы имен, перечисленные в @EXPORT и @EXPORT_OK.

С точки зрения импортирующего пакета, директива use задает список импортируемых имен, группу, указанную в %EXPORT_TAGS, шаблон имен или вообще ничего – в последнем случае в программу импортируются имена, указанные в @EXPORT.

Для импорта имен из модуля Bestiary можно включить любую из следующих инструкций:

```
use Bestiary;                    # Импортировать имена из @EXPORT
use Bestiary ();                 # Ничего не импортировать
use Bestiary qw(ram @llama);     # Импортировать функцию ram и массив @llama
use Bestiary qw(:camelids);      # Импортировать $camel и @llama
use Bestiary qw(:DEFAULT);       # Импортировать имена @EXPORT
use Bestiary qw(/am/);           # Импортировать $camel, @llama и ram
use Bestiary qw(/^\$/);          # Импортировать все скаляры
use Bestiary qw(:critters !ram); # Импортировать critters, но не ram
use Bestiary qw(:critters !:camelids); # Импортировать critters, за исключением camelids
```

Отсутствие элемента в списке экспорта (или явное удаление из списка импорта с помощью восклицательного знака) не делает его недоступным программе, использующей модуль. Программа всегда может обратиться к содержимому пакета модуля, полностью квалифицировав его именем пакета, например, %Bestiary::gecko. (Поскольку лексические переменные не принадлежат пакетам, возможность закрытия сохраняется; см. раздел «Закрытые методы» в следующей главе.)

Чтобы увидеть, как обрабатываются спецификации и что в действительности импортирует ваш пакет, можно сказать BEGIN { $Exporter::Verbose=1 }.

Exporter является модулем Perl, и, если вам интересно, вы можете увидеть, какие фокусы он выделывает с typeglob для экспорта имен из одного пакета в другой. Ключевой функцией внутри Exporter служит функция import, которая назначает псевдонимы, делая возможным обращение к имени из одного пакета в другом. На практике команда use Bestiary *LIST* совершенно эквивалентна следующим строкам:

```
BEGIN {
    require Bestiary;
    import Bestiary LIST;
}
```

Это значит, что вашим модулям не обязательно обращаться к Exporter. При загрузке модуль может делать все, что угодно, ведь use просто вызывает для модуля обычный метод import, а вы можете определить этот метод так, чтобы он делал все что вам угодно.

Экспортирование без вызова метода import модуля Exporter

Exporter определяет метод с именем export_to_level, применяемый, когда (по какой-то причине) нельзя непосредственно вызвать метод import модуля Exporter. Метод export_to_level вызывается так:

```
MODULE->export_to_level($where_to_export, @what_to_export);
```

где $where_to_export – целое число, определяющее, насколько далеко в стек вызовов должны экспортироваться ваши имена, а @what_to_export – массив с экспортируемыми именами (обычно @_).

Предположим, например, что в модуле Bestiary есть своя функция import:

```
package Bestiary;
@ISA = qw(Exporter);
@EXPORT_OK = qw ($zoo);

sub import {
    $Bestiary::zoo = "зверинец";
}
```

Наличие этой функции import не позволяет наследовать функцию import из Exporter. Чтобы функция import из Bestiary, после того как она установит $Bestiary::zoo, вела себя точно так же, как функция import из Exporter, ее следует определить так:

```
sub import {
    $Bestiary::zoo = "зверинец";
    Bestiary->export_to_level(1, @_);
}
```

При этом имена экспортируются на один уровень «выше» текущего пакета, т.е. в любую программу или модуль, которые используют Bestiary.

Однако, если вам требуется лишь это, вероятно, нет смысла наследовать весь модуль Exporter. Достаточно импортировать его метод import:

```
package Bestiary;
use Exporter qw(import); # v5.8.3 и выше
```

Проверка версий

Если в модуле определена переменная $VERSION, программа, использующая этот модуль, сможет проверить, достаточно ли свежей является его версия. Например:

```
use Bestiary 3.14;   # Bestiary должен иметь версию 3.14 или выше
use Bestiary v1.0.4; # Bestiary должен иметь версию 1.0.4 или выше
```

Эти инструкции становятся вызовами метода Bestiary->VERSION, который наследуется из модуля UNIVERSAL (см. главу 12).

Директива require также позволяет проверять версию вызовом метода VERSION:

```
require Bestiary;
Bestiary->VERSION( '2.71828' );
```

Версии модулей, увы, устроены сложнее, чем хотелось бы, и это неизбежно по историческим причинам. Первоначально номера версий назначались без оглядки на возможный анализ $VERSION. Это могло быть число, строка или результат некоторой операции. Долгие годы не существовало стандартного формата строк с номерами версий, поэтому люди выдумывали самые необычные номера версий, такие как "1.23alpha".[1] Следующие номера версий считаются эквивалентными:

```
our $VERSION = 1.002003;
our $VERSION = '1.002003';
our $VERSION = v1.2.3;

use version;
our $VERSION = version->new( "v1.2.3" );
```

Такая схема никому не мешала, пока Perl не изменил собственную схему нумерации версий между версиями 5.005 и v5.6. Теперь номером версии стала *v-строка*, специальный литерал, представляющий номер версии и допускающий произвольное количество точек. Фактически эти *v-строки* были целыми числами, упакованными в символы. Затем разработчикам Perl пришла идея создать объект, представляющий версию, и модуль version. Если вы вынуждены поддерживать по-настоящему древние версии Perl (сначала выразим наше сочувствие: v5.6 появилась еще в прошлом тысячелетии), старайтесь придерживаться простых последовательностей.

Perl предполагает, что часть после десятичной точки состоит из трех знаков, из-за чего результаты сравнения могут показаться странными. Так, версия 1.9 предшествует версии 1.10, даже при том, что лексикографически .9 сортируется после .1. Perl воспринимает эти числа как 1.009 и 1.010. Должны ли вы с этим соглашаться? Нет. Придется ли вам с этим смириться? Да. (Но, как бы то ни было, используйте формат v1.9, где это возможно, так как он будет поддерживаться в будущем.)

И это еще не все. Появилось соглашение для версий, находящихся в разработке. Оно предписывает добавлять в номер версии знак подчеркивания (_) или окончание -TRIAL. Многие инструменты CPAN считают такие модули нестабильными. Это позволяет авторам выгружать в CPAN промежуточные версии для опробования тестировщиками и предварительные версии для опробования пользователями, не вынуждая никого использовать не готовые к выпуску версии (см. главу 19).

```
our $VERSION = '1.234_001'
```

Кавычки необходимы, чтобы сохранить символ подчеркивания в литерале, который в противном случае будет выброшен анализатором, поскольку компилятор допускает использование этого символа в числовых литералах.

Дэвид Голден (David Golden) подробнее рассказывает об этом в статье «Version numbers should be boring» (Номера версий должны быть простыми) (*http://www.dagolden.com/index.php/369/version-numbers-should-be-boring/*).

Обратите внимание, что в последних версиях Perl можно избавиться от объявления our и просто писать:

```
package Bestiary v1.2.3;
```

---

[1] Чтобы осознать, насколько бредовыми могут быть номера версий, загляните в модуль CPAN::DistnameInfo, реализующий распознавание версий.

Действия при обнаружении неизвестных имен

В некоторых ситуациях может потребоваться *воспрепятствовать* экспорту некоторых имен. Обычно это относится к модулям, содержащим функции или константы, не имеющие смысла в некоторых системах. Можно помешать модулю Exporter экспортировать эти имена, поместив их в массив @EXPORT_FAIL.

Когда программа попытается импортировать одно из этих имен, Exporter даст модулю возможность обработать ситуацию, прежде чем сгенерировать ошибку. При этом он передаст список ошибочных имен методу export_fail, который можно определить так (в предположении, что ваш модуль использует модуль Carp):

```
use Carp;
sub export_fail {
    my $class = shift;
    carp "К сожалению, эти имена недоступны: @_";
    return @_;
}
```

Exporter предоставляет вызываемый по умолчанию метод export_fail, который просто возвращает список в неизменном виде и прерывает вызов use, возбуждая исключительную ситуацию для каждого имени. Если export_fail возвращает пустой список, значит, ошибки отсутствуют, и все запрошенные имена экспортируются.

Вспомогательные функции для обработки групп имен

Поскольку имена, перечисленные в %EXPORT_TAGS, должны также присутствовать в @EXPORT или @EXPORT_OK, в Exporter есть две функции, позволяющие добавить эти помеченные группы имен:

```
%EXPORT_TAGS = (foo => [qw(aa bb cc)], bar => [qw(aa cc dd)]);

Exporter::export_tags("foo");     # добавить aa, bb и cc в @EXPORT
Exporter::export_ok_tags("bar"); # добавить aa, cc и dd в @EXPORT_OK
```

Попытка передать имя, не являющееся идентификатором группы имен, вызовет ошибку.

Замещение встроенных функций

Многие встроенные функции могут быть *замещены* (*overridden*), однако делать это стоит лишь в редких случаях, имея достаточные на то основания. Обычно замена может осуществляться в пакете, пытающемся эмулировать отсутствующие встроенные функции в системе, отличной от UNIX. (Не путайте замещение (overriding) с *перегрузкой* (*overloading*), которая придает встроенным операторам дополнительные объектно-ориентированные значения, но ничего не замещает. Дополнительно читайте об этом в описании модуля overload в главе 13.)

Замещение может осуществляться только путем импортирования имени из модуля – обычного предварительного объявления для этого недостаточно. Если говорить совсем откровенно, замещение – это сохранение ссылки на код в записи typeglob, как в выражении *open = \&myopen. Кроме того, такое присваивание должно осуществляться в другом пакете; это сделано с целью затруднить случайное замещение при создании псевдонима typeglob. Если вам действительно нужно

выполнить собственное замещение, не отчаивайтесь, потому что директива subs позволяет заранее, посредством импортирующего синтаксиса, объявить имена подпрограмм, которые впоследствии заместят встроенные:

```
use subs qw(chdir chroot chmod chown);
chdir $somewhere;
sub chdir { ... }
```

Вообще говоря, модулям не следует экспортировать такие встроенные имена, как open или chdir, в составе списка по умолчанию (@EXPORT), поскольку они могут попасть в другое пространство имен и неожиданным образом изменить семантику. Если вместо этого включить такое имя в список @EXPORT_OK, импортерам придется запрашивать замещение встроенного имени явным образом, и все останутся честными друг перед другом.

Исходные версии встроенных функций всегда доступны через псевдопакет CORE. Поэтому CORE::chdir всегда вызовет версию, изначально встроенную в Perl, даже если заместить ключевое слово chdir.

Ну, почти всегда. Описанный механизм замещения умышленно ограничен тем пакетом, который запрашивает импорт. Но есть механизм с более широким спектром действия, позволяющий заместить встроенную функцию везде, игнорируя границы пространств имен. Это достигается путем определения функции в псевдопакете CORE::GLOBAL. В следующем примере оператор glob заменяется на нечто, что понимает регулярные выражения. (Учтите, что этот пример реализует не все, что нужно для чистого замещения встроенного оператора glob, который ведет себя по-разному в скалярном и списочном контекстах. На практике многие встроенные функции Perl имеют такие зависящие от контекста режимы, и каждое правильно написанное замещение должно их адекватным образом поддерживать. Полнофункциональный пример замещения glob можно найти в модуле File::Glob, поставляемом с Perl.) Тем не менее, вот асоциальная версия:

```
*CORE::GLOBAL::glob = sub {
    my $pat = shift;
    my @got;
    local *D;
    if (opendir D, ".") {
        @got = grep /$pat/, readdir D;
        closedir D;
    }
    return @got;
}

package Whatever;

print <^[a-z_]+\.pm\$>; # показать все директивы в текущем каталоге
```

При глобальном замещении glob происходит вытесняющее навязывание нового (и подрывного) режима действия оператора glob в каждом пространстве имен без уведомления или согласия модулей, являющихся владельцами этих пространств имен. Конечно, это нужно делать с крайней осторожностью – если делать вообще. Скорее всего, так делать не стоит.

Наша философия замещения такова: приятно быть важным, но важнее быть приятным.

12

Объекты

Прежде всего, вам потребуется понимание пакетов и модулей; см. главу 10 и главу 11. Помимо этого – понимание ссылок и структур данных; см. главу 8 и главу 9. Не лишним будут некоторые познания в объектно-ориентированном программировании (ООП), поэтому в следующем разделе мы прочтем вам небольшой курс ООЖ (объектно-ориентированного жаргона).

Объектно-ориентированная модель Perl, вероятно, существенно отличается от моделей ООП, с которыми вы работали в других языках. Читая эту главу, лучше забыть все, что вы знаете из этих других языков.

Краткая памятка по объектно-ориентированному жаргону

Объект – это структура данных с некоторым набором поведенческих характеристик (behaviors). Обычно мы говорим о поведении объекта, как если бы оно осуществлялось им непосредственно, иногда доходя до антропоморфизма. Например, можно услышать, что прямоугольник «умеет» выводить себя на экран или что он «знает», как вычислить свою площадь.

Каждый объект обладает своим поведением благодаря тому, что является *экземпляром* (*instance*) *класса*. Класс определяет *методы*: элементы поведения, действующие для класса и его экземпляров. Когда это существенно, мы называем методы, применимые только для конкретного объекта, *методами объекта* или *методами экземпляра*, а методы, применимые для класса в целом, *методами класса*. Так принято говорить, однако для Perl метод – всегда только метод, и Perl различает методы лишь по типам первого аргумента.

Метод экземпляра можно рассматривать как некоторое действие, осуществляемое конкретным объектом, например вывод самого себя, копирование себя или изменение одного или нескольких своих свойств («установить название этого меча равным Андурил»). Методы класса могут осуществлять операции над совокупностью многих объектов («вывести названия всех мечей») или выполнять другие действия,

не зависящие от конкретных объектов («с данного момента, когда выковывается новый меч, регистрировать его владельца в этой базе данных»). Методы, создающие экземпляры (объекты) класса, называются *конструкторами* («создать меч с рукояткой, украшенной драгоценными камнями и таинственной надписью»). Обычно это методы класса («сделать мне новый меч»), но могут быть и методы объекта («сделать точную копию этого меча»).

Класс может *наследовать* (*inherit*) методы *родительских классов* (*parent classes*), называемых также *базовыми классами* (*base classes*) или *надклассами* (*superclasses*). В этом случае класс называется *производным классом* (*derived class*) или *подклассом* (*subclass*). (Еще более запутывает то обстоятельство, что под «базовым классом» в литературе иногда имеется в виду «самый верхний» надкласс. Мы в таком значении его не используем.) Наследование придает новому классу элементы поведения существующего класса, но позволяет изменять или добавлять новые элементы поведения, отсутствующие у его родителей. При вызове метода, определения которого нет в классе, Perl автоматически ищет определение в родительских классах. Например, класс меча может наследовать свой метод `attack` от общего класса клинка. У родителей могут быть свои родители, и при необходимости Perl выполнит поиск в классах более далеких предков. Класс клинка, в свою очередь, может наследовать метод `attack` от еще более общего класса оружия.

Когда вызывается метод `attack` объекта, итоговое поведение может зависеть от того, является ли объект мечом или стрелой. Возможно, никакого различия вообще не будет, например, если и меч, и стрела унаследовали свой метод нанесения ущерба от более общего класса «оружие». Но если различие в поведении есть, механизм диспетчеризации методов выберет метод `attack`, подходящий для рассматриваемого объекта. Полезное свойство, позволяющее всегда выбрать наиболее подходящее поведение для конкретного типа объекта, называется *полиморфизмом*. Это важный вид безразличия.

При реализации класса необходимо заботиться о внутреннем содержании своих объектов, но при использовании класса его объекты следует считать черными ящиками. Невозможно увидеть, что находится внутри, не требуется знать, как он работает, – вы взаимодействуете с ящиком лишь на его условиях: через методы, предоставляемые классом. Даже если известно, что конкретно эти методы делают с объектом, следует сопротивляться желанию покопаться в объекте напрямую. Это как пульт управления телевизором: даже если вы знаете, что происходит внутри него, не следует копаться в его внутренностях из праздного любопытства.

Perl позволяет заглянуть внутрь объекта из-за пределов класса. Но при этом нарушается принцип *инкапсуляции*, согласно которому любой доступ к объекту должен осуществляться только через его методы. Инкапсуляция отделяет опубликованный интерфейс (описывающий, как объект должен использоваться) от реализации (определяющей, как он в действительности работает). В Perl отсутствуют явные ограничения помимо этого неписаного контракта между разработчиком и пользователем. Ожидается, что обе стороны проявят здравый смысл и будут соблюдать приличия: пользователь полагается на документированный интерфейс, разработчик обещает его не изменять.

Perl не требует, чтобы вы придерживались определенного стиля программирования, и не одержим идеей закрытости, как некоторые другие объектно-ориентированные языки программирования. Однако Perl одержим идеей свободы, и одна из

свобод, которые Perl предоставляет программисту, это право избрать такую меру закрытости, какую он пожелает. На практике Perl способен обеспечить более сильную закрытость классов и объектов, чем C++. Это означает, что Perl ни в чем не ограничивает программиста и, в частности, не ограничивает его право ограничивать самого себя (если вас привлекают такого рода вещи). В разделах «Закрытые методы» и «Использование замыканий в закрытых объектах», далее в этой главе, рассказывается, как можно ужесточить самодисциплину.

Конечно, это далеко не все, что можно рассказать по теме объектов или подходов к объектно-ориентированному программированию. Но это не входит в наши задачи. Поэтому продолжим.

Система объектов Perl

Perl не имеет специального синтаксиса определения объектов, классов или методов. Для реализации этих трех понятий применяются существующие конструкции.[1] Вот несколько простых определений, которые могут показаться вам ободряющими:

Объект является просто ссылкой... э-э, объектом ссылки.

Поскольку ссылки позволяют отдельным скалярам представлять сложные совокупности данных, не должно вызывать удивления, что для представления объектов используются ссылки. Технически сам объект не является ссылкой – это то, на что указывает ссылка. Однако программисты на Perl часто не вникают в эту тонкость, и, поскольку мы считаем это прекрасной метонимией, то продолжим использовать ее, когда нам это будет удобно.[2]

Класс является просто пакетом.

Пакет служит классом благодаря использованию его подпрограмм для выполнения методов класса и использованию переменных пакета для хранения глобальных данных класса. Часто один или несколько классов хранят в модуле

Метод является просто подпрограммой.

Вы просто объявляете подпрограммы в пакете, используемом в качестве класса, и они будут использоваться в качестве методов класса. Вызов метода – новый способ обращения к подпрограммам – передает дополнительный аргумент: объект или пакет, используемый для вызова метода.

Вызов методов

Если бы потребовалось выделить квинтэссенцию объектно-ориентированного программирования, ею стало бы понятие *абстракции* – единая тема, связывающая все эти сложные слова, которыми энтузиасты ООП любят перебрасываться, такие как полиморфизм, наследование и инкапсуляция. Мы верим в эти причудливые слова, но будем рассматривать их с практической позиции: что они означают для вызова методов? Методы формируют ядро системы объектов, поскольку

---

[1] А вот это классный пример повторного использования кода!

[2] Мы предпочитаем лингвистическую живость математической строгости. Можете не соглашаться.

представляют уровень абстракции, необходимый для реализации всех этих причудливых терминов. Вместо непосредственного обращения к данным в объекте вызывают метод экземпляра. Вместо непосредственного обращения к подпрограмме в каком-то пакете вызывают метод класса. Располагая этот уровень косвенности между использованием класса и реализацией класса, разработчик программы получает возможность свободно изменять внутреннее устройство класса, не особо рискуя нарушить работу программ, которые его используют.

Perl поддерживает две синтаксические формы вызова методов. В одной используется уже знакомый вам стиль Perl, а вторую вы могли встречать в других языках программирования. Какая бы форма вызова метода ни использовалась, подпрограмме-методу всегда передается дополнительный начальный аргумент. Если для вызова метода используется класс, этот аргумент будет именем класса. А если объект, этот аргумент будет ссылкой на объект. Чем бы он ни был, мы будем называть его *инвокантом* (*invocant*)[1] метода. Для метода класса инвокантом служит имя пакета. Для метода экземпляра – ссылка на объект.

Иными словами, инвокант – это то, с *чем* вызывается метод. Иногда в литературе по ООП его называют *агентом* (*agent*) или *актором* (*actor*). Грамматически инвокант не является ни субъектом действия, ни получателем этого действия. Он более похож на косвенное дополнение, бенефициара, от имени и по поручению которого осуществляется действие, – как слово «мне» в команде «выкуй мне меч!». Семантически можно считать инвокант либо тем, что создает вызов, либо тем, что служит предметом вызова, – в зависимости от того, что больше подходит вашему складу ума. Мы не собираемся учить вас, как следует думать. (Во всяком случае, об этом.)

Обычно методы вызываются явно, но могут вызываться и неявно, деструкторами объектов, перегруженными операторами или связанными переменными. Строго говоря, это не обычные вызовы подпрограмм, а вызовы методов, автоматически выполняемые от имени объекта. Деструкторы описываются далее в этой главе, перегрузка – в главе 13, а связанные переменные – в главе 14.

Одно из отличий методов от подпрограмм состоит в моменте времени, когда выполняется поиск их пакетов, т.е. насколько рано (или поздно) Perl решает, какой код должен быть выполнен в результате вызова метода или подпрограммы. Поиск пакета подпрограммы производится во время компиляции, прежде чем программа начнет выполняться.[2] Напротив, поиск пакета метода не производится до момента фактического вызова метода. (Проверка прототипов происходит на этапе компиляции, поэтому обычные подпрограммы могут использовать прототипы, а методы – нет.)

Причина, почему пакет метода нельзя определить раньше, относительно проста: пакет определяется классом инвоканта, а инвокант неизвестен, пока метод не будет вызван фактически. В основе ООП лежит простая логическая цепочка: когда известен инвокант, известен класс инвоканта, а когда известен класс, известна

---

[1] От лат. in – в, внутрь и vocare – звать, призывать. – *Прим. ред.*

[2] Точнее говоря, вызов подпрограммы разрешается в некоторый typeglob, ссылка на который компилируется в дерево кодов операций. О значении этого typeglob можно договориться даже на этапе исполнения – именно это позволяет `AUTOLOAD` автоматически загружать подпрограммы. Обычно же и значение этого typeglob разрешается на этапе компиляции с помощью определения подпрограммы с надлежащим именем.

его цепочка наследования, а когда известна цепочка наследования, известна фактическая подпрограмма, которую нужно вызвать.

За использование алгоритмов абстракции приходится платить. Из-за позднего поиска методов объектно-ориентированное решение в Perl, скорее всего, будет работать медленнее, чем соответствующее решение без применения ООП. Для некоторых изощренных приемов, описываемых ниже, замедление может быть *значительным*. Однако многие распространенные задачи решаются написанием кода не быстрого, но умного. Вот тут и начинает блистать ООП.

Вызов методов с помощью оператора стрелки

Как уже упоминалось, есть два стиля вызова методов. Первый выглядит так:

```
INVOCANT->METHOD(LIST)
INVOCANT->METHOD
```

По очевидной причине этот стиль обычно называют стрелочной формой вызова. (Не путайте -> и =>, «двуствольную» стрелку, используемую вместо запятой.) Если есть аргументы, необходимы круглые скобки. При выполнении вызова Perl сначала отыскивает подпрограмму, определяемую совокупностью класса инвоканта *INVOCANT* и имени метода *METHOD*, а затем вызывает ее, передавая *INVOCANT* в качестве первого аргумента.

Если *INVOCANT* является ссылкой, мы говорим, что *METHOD* вызывается как метод экземпляра (объекта), а если *INVOCANT* является именем пакета, говорим, что *METHOD* вызывается как метод класса. На практике различий между тем и другим нет, за исключением того, что имя пакета более очевидным образом связано с самим классом, чем с объектами этого класса. Поверьте нам на слово, объекты тоже знают, к какому классу они принадлежат. Очень скоро мы расскажем, как связать объект с именем класса, но использовать объекты можно и без этой информации.

Например, чтобы создать объект с помощью метода класса `summon` и затем вызвать метод экземпляра `speak` с полученным объектом, можно сказать:

```
$mage = Wizard->summon("Gandalf"); # метод класса
$mage->speak("friend")             # метод экземпляра
```

Методы `summon` и `speak` определены классом `Wizard` (Волшебник) – или одним из классов, которые он наследует. Но вас это не должно беспокоить. Не вмешивайтесь в дела Волшебников.

Поскольку оператор стрелки левоассоциативен (см. главу 3), две инструкции можно даже объединить в одну:

```
Wizard->summon("Gandalf")->speak("friend");
```

Иногда требуется вызвать метод, не зная заранее его имени. Можно выбрать стрелочную форму вызова метода и заменить имя метода простой скалярной переменной:

```
$method = "summon";
$mage = Wizard->$method("Gandalf"); # Вызвать Wizard->summon

$travel = $companion eq "Shadowfax ? "ride"    walk";
$mage->$travel("seven leagues");    # Вызвать $mage->ride или $mage->walk
```

Хотя имя метода служит для его косвенного вызова, такое использование не запрещается в области действия прагмы `use strict 'refs'`, поскольку в действительности для всех обращений к методам их поиск осуществляется по таблицам имен в момент разрешения.

В нашем примере мы сохранили в `$travel` имя метода, но это могла быть и ссылка на подпрограмму. В этом случае минуется алгоритм поиска метода, но иногда это как раз то, что требуется. Посмотрите раздел «Закрытые методы» и обсуждение метода can в разделе «UNIVERSAL: первичный класс предков». О том, как создать ссылку на конкретный метод конкретного экземпляра, читайте в разделе «Замыкания» главы 8.

Вызов методов с использованием косвенных объектов

Второй стиль вызова методов выглядит так:

```
METHOD INVOCANT (LIST)
METHOD INVOCANT LIST
METHOD INVOCANT
```

Круглые скобки вокруг *LIST* не обязательны; если их опустить, метод действует как списочный оператор. Поэтому можно создавать такие инструкции, как показано ниже, с использованием данного стиля вызова методов. Обратите внимание на отсутствие точки после имени класса или экземпляра:

```
no feature "switch";   # включить более снисходительный режим (см. ниже)
$mage = summon Wizard "Gandalf";
$nemesis = summon Balrog home => "Moria", weapon => "whip";
move $nemesis "bridge";
speak $mage "Ты не пройдешь";
break $staff;          # надежнее использовать: break $staff ();
```

Синтаксис списочного оператора должен быть вам знаком; это тот же стиль, который используется для передачи дескрипторов файлов в `print` и `printf`:

```
print STDERR "help!!!\n";
```

Это аналогично таким английским предложениям, как «Give Gollum the Preciousss»[1], поэтому мы называем это формой *косвенного объекта* (*indirect object*). Ожидается, что инвокант находится в *позиции косвенного объекта* (*indirect object slot*). Когда вы читаете, что встроенной функции, например `system` или `exec`, что-то передается в ее «позицию косвенного объекта», это означает, что передается этот дополнительный аргумент без запятых в той же позиции, что при вызове метода с использованием синтаксиса косвенного объекта.

Форма косвенного объекта даже допускает использовать в качестве INVOCANT конструкцию *BLOCK*, вычисляемую в значение-объект (ссылку) или в значение-класс (пакет). Это позволяет так объединить два приведенных вызова в одной инструкции:

```
speak { summon Wizard "Gandalf" } "friend";
```

[1] Тот, кто произносит эту фразу (а именно Gollum), говорит о себе в третьем лице. – *Прим. ред.*

Синтаксическая путаница с косвенными объектами

Один синтаксис часто оказывается более удобочитаемым, чем другой. Синтаксис косвенного объекта проще, но страдает некоторыми формами синтаксической неоднозначности. Во-первых, *LIST* в форме вызова косвенного объекта обрабатывается синтаксическим анализатором так же, как любой другой списочный оператор. Поэтому считается, что в выражении

```
enchant $sword ($pips + 2) * $cost;
```

скобки содержат все аргументы независимо от того, что следует за ними. В результате это эквивалентно следующему выражению:

```
($sword->enchant($pips + 2)) * $cost;
```

Маловероятно, что такое выражение даст нужный нам результат: `enchant` вызывается только с `$pips + 2`, а значение, возвращаемое методом, умножается затем на `$cost`. Как и с другими списочными операторами, также следует проявлять осторожность в отношении старшинства `&&` и `||` относительно `and` и `or`.

Например, вызов:

```
name $sword $oldname || "Glamdring"; # здесь нельзя использовать "or"!
```

интерпретируется как подразумевавшееся:

```
$sword->name($oldname || "Glamdring");
```

но:

```
speak $mage "friend" && enter(); # здесь должен быть "and"!
```

становится двусмысленным:

```
$mage->speak("friend" && enter());
```

Это можно исправить, переписав код в одной из следующих эквивалентных форм:

```
enter() if $mage->speak("friend");
$mage->speak("friend") && enter();
speak $mage "friend" and enter();
```

Вторая синтаксическая проблема формы косвенного объекта состоит в том, что в ней *INVOCANT* может быть только именем, скалярной переменной без индекса или блоком.[1] Как только анализатор встречает один из этих вариантов, то считает его за *INVOCANT*, а потому начинает поиск списка *LIST*. Значит, вызовы:

```
move $party->{LEADER}; # вероятно, ошибочно!
move $riders[$i];      # вероятно, ошибочно!
```

в действительности интерпретируются как:

```
$party->move->{LEADER};
$riders->move([$i]);
```

хотя, на самом деле, вероятно, требовалось:

---

[1] Внимательные читатели вспомнят, что это в точности тот список синтаксических элементов, которые допускаются после разыменовывающего символа при разыменовании переменных, например `@ary`, `@$aryref` или `@{$aryref}`.

```
$party->{LEADER}->move;
$riders[$i]->move;
```

Анализатор заглядывает вперед лишь немного, чтобы найти инвокант для косвенного объекта, т.е. даже не так далеко, как при поиске для унарного оператора. В первом варианте таких странностей не возникает, поэтому вы, возможно, предпочтете оператор-стрелу в качестве оружия.

Даже в английском языке есть аналогичная проблема. Возьмите команду: «Throw your cat out the window a toy mouse to play with». Если поторопиться с разбором этого предложения, в окно полетит кошка, а не мышка (если только вы не заметите, что кошка уже за окном).[1] Подобно Perl, английский язык предлагает два варианта синтаксиса для обозначения агента: «Throw your cat the mouse» и «Throw the mouse to your cat». Иногда более длинная форма оказывается понятнее и проще, а иногда – более короткая. В Perl, по крайней мере, вы обязаны заключить сложный косвенный объект в фигурные скобки.

Классы, цитирующие имя пакета

Последняя синтаксическая двусмысленность вызова методов в стиле косвенных объектов состоит в том, что они вообще могут быть опознаны не как обращение к методу, если в текущем пакете есть подпрограмма с таким же именем, как у метода. В случае вызова метода класса, имя пакета которого совпадает с именем инвоканта, избавиться от этой неопределенности при сохранении синтаксиса косвенного объекта можно, добавив два двоеточия к имени класса, цитируя (package-quote), тем самым, имя пакета:

```
$obj = method CLASS::  # принудительно интерпретируется как "CLASS"->method
```

Это важно, потому что обычная запись:

```
$obj = new CLASS;      # может быть воспринято не как вызов метода
```

не всегда ведет себя как нужно, если в текущем пакете есть подпрограмма с именем new или CLASS. Даже если старательно использовать для вызова метода стрелку, а не косвенный объект, в редких случаях все же могут возникать проблемы. Ценой помех, вызываемых лишними знаками пунктуации, обозначение CLASS:: гарантирует правильный анализ вызова метода. Первые два примера из приведенных ниже не всегда анализируются одинаково, в отличие от следующих двух:

```
$obj = new ElvenRing;       # может быть воспринято как new("ElvenRing")
                            # или даже new(ElvenRing())
$obj = ElvenRing->new;      # может быть воспринято как ElvenRing()->new()

$obj = new ElvenRing::;     # всегда "ElvenRing"->new()
$obj = ElvenRing::->new;    # всегда "ElvenRing"->new()
```

Эту нотацию можно сделать симпатичнее посредством творческого выравнивания:

---

[1] На русском языке команда могла бы выглядеть так: «Бросьте кошке за окном игрушечную мышку, пусть позабавится». Естественно, в основном тексте речь идет об оригинале. Читая перевод команды, можно только сделать вид, что не очень понятно, кто же будет забавляться. – *Прим. ред.*

```
$obj = new ElvenRing::
            name     => "Narya",
            owner    => "Gandalf",
            domain   => "fire",
            stone    => "ruby";
```

Все еще можно сказать: «Фу!», глядя на это удвоенное двоеточие, поэтому сообщим, что всегда можно обойтись простым именем класса, если соблюдаются два условия. Во-первых, если в классе нет подпрограммы с именем, совпадающим с именем класса. (Если следовать соглашению, требующему, чтобы имена подпрограмм, как new, начинались символами нижнего регистра, а имена классов, как ElvenRing, – символами верхнего регистра, проблема исчезнет.) Во-вторых, если класс загружен одной из следующих инструкций:

```
use ElvenRing;
require ElvenRing;
```

Наличие любого из этих объявлений гарантирует, что ElvenRing будет интерпретироваться как имя модуля, а любое голое имя вроде new перед именем класса ElvenRing – как вызов метода, даже если в текущем пакете будет определена подпрограмма new. Обычно проблем с косвенными объектами не возникает, если не пытаться втиснуть много классов в один файл, из-за чего Perl может не понять, что некоторое имя пакета должно восприниматься как имя класса. Тех, кто дает своим подпрограммам имена типа ModuleNames, тоже поджидают неприятности.

Именно это (почти) произошло с нами в примере, где мы сказали:

```
no feature "switch";
```

Если предположить, что была использована рекомендованная директива use v5.14 или аналогичная, с версией v5.10 или выше, в языке появится новое ключевое слово break, используемое в конструкции given. Мы отключили особенность "switch", потому что иначе компилятор будет считать имя break ключевым словом. Добавление круглых скобок в конце не решает проблему, даже при том, что обычно это действенное решение (или *должно быть* таковым), обеспечивающее однозначную интерпретацию такого синтаксиса вызова метода. Компилятор оказывается в растерянности, не зная, *что* с этим делать, и это мешает ему продолжить свою работу.

Создание объектов

Все объекты являются ссылками, но не все ссылки являются объектами. Ссылка не может действовать как объект, пока объект ссылки не будет помечен особой меткой, сообщающей, к какому пакету он принадлежит. Маркировка объекта ссылки именем пакета, и, следовательно, именем класса, так как класс – это просто пакет, заключается в применении к ссылке функции bless и называется *blessing* («благословение»). Можете считать, что в результате ссылка превращается в объект, хотя точнее было бы сказать, что ссылка превращается в ссылку на объект.

Функция bless принимает один или два аргумента. Первый аргумент является ссылкой, а второй – пакетом, которым помечается объект ссылки. Если второй аргумент опущен, используется текущий пакет.

```
$obj = { };              # Получить ссылку на анонимный хеш.
bless($obj);             # Сделать хеш объектом текущего пакета.
bless($obj, "Critter");  # Сделать хеш объектом класса Critter
```

Здесь мы использовали ссылку на анонимный хеш, который обычно используется как структура данных для объектов. Хеши являются очень гибкой конструкцией. Но хотелось бы подчеркнуть, что функцию bless можно применить к ссылке, указывающей на любой элемент, на который вообще можно ссылаться, в том числе скаляры, массивы, подпрограммы и typeglob. Функцию bless можно применить даже к хешу таблицы имен пакета, если найдется уважительная причина. (И даже если не найдется.) Объектная ориентированность в Perl совершенно ортогональна структуре данных.

После того как объект ссылки будет помечен, встроенная функция ref для такой ссылки будет возвращать не встроенный тип, а имя класса, например HASH. Чтобы получить встроенный тип, используйте функцию reftype из модуля attributes. См. описание attributes в главе 29.

Вот так и создается объект. Берется ссылка, вызовом функции bless помечается именем пакета, и все. Этого достаточно, чтобы создать минимальный класс. Если вы лишь пользуетесь классом, то нужно еще меньше, потому что разработчик класса спрячет вызов bless в методе-*конструкторе*, который создает и возвращает экземпляры класса. Поскольку bless возвращает свой первый аргумент, обычный конструктор может быть очень прост:

```
package Critter;
sub spawn { bless {}; }
```

Или, если расписать подробнее:

```
package Critter;
sub spawn {
    my     $self = {};          # Ссылка на пустой анонимный хеш
    bless  $self, "Critter";    # Сделать этот хеш объектом Critter
    return $self;               # Вернуть только что созданный объект Critter
}
```

Имея такое определение, объект Critter можно создать вызовом:

```
$pet = Critter->spawn;
```

Наследуемые конструкторы

Как и все методы, конструктор – это просто подпрограмма, но мы не вызываем его как подпрограмму. Мы всегда вызываем его как метод – метод класса, в данном случае, потому что инвокантом является имя пакета. Вызовы методов имеют два отличия от вызовов обычных подпрограмм. Во-первых, они получают дополнительный аргумент, о чем говорилось выше. Во-вторых, они подчиняются наследованию, позволяющему одному классу использовать методы другого.

Механику наследования мы обсудим более строго в следующем разделе, а пока приведем несколько простых примеров, которые помогут вам в создании собственных конструкторов. Допустим, что есть класс Spider (паук), наследующий методы класса Critter (живое существо). В частности, предположим, что в классе Spider нет собственного метода spawn. Тогда при вызове этого метода применяются соответствия, перечисленные в табл. 12.1:

Таблица 12.1. Соответствие методов подпрограммам

Вызов метода	Итоговый вызов подпрограммы
Critter->spawn()	Critter::spawn("Critter")
Spider->spawn()	Critter::spawn("Spider")

В обоих случаях вызывается одна и та же подпрограмма, но с разными аргументами. Обратите внимание, что конструктор spawn, приведенный выше, игнорирует свой аргумент, а это значит, что объект Spider будет ошибочно «благословлен» в класс Critter. Более правильный конструктор должен передать функции bless имя пакета (полученное в первом аргументе):

```
sub spawn {
    my $class = shift;      # Сохранить имя пакета
    my $self  = { };
    bless($self, $class);   # "Благословить" ссылку в этот пакет
    return $self;
}
```

Теперь в обоих случаях можно использовать одну и ту же подпрограмму:

```
$vermin = Critter->spawn;
$shelob = Spider->spawn;
```

И каждый объект будет принадлежать правильному классу. Выбор правильного класса гарантируется даже при косвенном вызове:

```
$type   = "Spider";
$shelob = $type->spawn;     # То же, что и "Spider"->spawn
```

Это по-прежнему метод класса, а не экземпляра, потому что его инвокант содержит строку, а не ссылку.

Будь переменная $type объектом, а не именем класса, текущее определение конструктора не работало бы, потому что функция bless требует имени класса. Но для многих классов есть смысл использовать существующий объект в качестве шаблона для создания нового объекта. В таких случаях можно создавать конструкторы, работающие и с объектами, и с именами классов:

```
sub spawn {
    my $invocant = shift;
    my $class    = ref($invocant) || $invocant; # Объект или имя класса
    my $self     = { };
    bless($self, $class);
    return $self;
}
```

Инициализаторы

Большинство объектов хранит внутреннюю информацию, которая косвенно обрабатывается методами объекта. До сих пор все наши конструкторы создавали пустые хеши, но нет причин оставлять их пустыми. Например, конструктор мог бы принимать дополнительные аргументы и хранить их в хеше парами вида ключ/значение. В литературе по ООП такие данные часто называются *свойствами*, *атрибутами*, *методами доступа*, *данными-членами*, *данными экземпляра*

и *переменными экземпляра* (*properties*, *attributes*, *accessors*, *member data*, *instance data*, *instance variables*). Более подробно атрибуты обсуждаются в разделе «Переменные экземпляра», далее в этой главе.

Представим себе класс Horse (лошадь) с атрибутами «name» (кличка) и «color» (масть):

```
$steed = Horse->new(name => "Shadowfax", color => "white");
```

Если объект реализован как ссылка на хеш, пары ключ/значение можно интерполировать непосредственно в хеш, как только инвокант будет удален из списка аргументов:

```
sub new {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    my $self = { @_ };          # Оставшиеся аргументы становятся атрибутами
    bless($self, $class);       # "Благословение" в объекты
    return $self;
}
```

На этот раз в качестве конструктора класса мы использовали метод с именем new, что может создать у программистов на языке C++ ложное ощущение понимания происходящего. Для Perl слово «new» не имеет специального значения; конструкторы можно называть как угодно. Любой метод, создающий и возвращающий объект, является конструктором de facto. В целом, мы советуем давать конструкторам осмысленные в контексте решаемой задачи имена. Например, конструкторы в модуле Tk имеют имена, соответствующие создаваемым графическим объектам. В модуле DBI конструктор connect возвращает объект – дескриптор базы данных, а другой конструктор, prepare, вызывается как метод экземпляра и возвращает объект – дескриптор команды. Но если нет подходящего в данном контексте имени конструктора, new станет не самым ужасным выбором. Ну, а если выбрать какое-то случайное имя, это заставит пользователей прочесть контракт интерфейса (т.е. документацию класса), прежде чем использовать его конструкторы.

Далее, можно оснастить свой конструктор парами ключ/значение по умолчанию, которые пользователь сможет переопределить с помощью аргументов:

```
sub new {
    my $invocant = shift;
    my $class    = ref($invocant) || $invocant;
    my $self = {
        color  => "гнедой",
        legs   => 4,
        owner  => undef,
        @_,                 # Заместить прежние атрибуты
    };
    return bless $self, $class;
}

$ed       = Horse->new;                      # 4-ногая гнедая лошадь
$stallion = Horse->new(color => "черный");  # 4-ногая вороная лошадь
```

При использовании в качестве метода экземпляра конструктор Horse игнорирует атрибуты инвоканта. Можно было бы создать второй конструктор, предназначенный для вызова в качестве метода экземпляра, и сделать так, чтобы значения

вызывающего объекта принимались как значения по умолчанию для атрибутов нового объекта:

```
$steed = Horse->new(color => "серовато-коричневый");
$foal  = $steed->clone(owner => "EquuGen Guild, Ltd.");

sub clone {
    my $model = shift;
    my $self  = $model->new(%$model, @_);
    return $self;          # Ранее "благословленный" с помощью ->new
}
```

(Эти инструкции можно поместить прямо в new, но тогда имя будет не вполне соответствовать функции.)

Обратите внимание, что даже в конструкторе clone мы не кодируем жестко имя класса Horse. Исходный объект, чем бы он ни был, вызывает свой собственный метод new. Если бы вместо $model->new мы написали Horse->new, этот класс не позволил бы наследовать его классам Зебра или Единорог. Не хотелось бы, клонируя Пегаса, получить лишь лошадь другой масти.

Иногда, однако, возникает противоположная проблема: вместо того, чтобы делить один конструктор между несколькими классами, желательно иметь несколько конструкторов в одном классе, например, чтобы вызвать конструктор базового класса перекладывая на него часть работы. Perl создает объекты без учета положения их классов в иерархии, т.е. он не вызывает автоматически конструкторы (или деструкторы) базовых классов, поэтому вашему конструктору придется сделать это самостоятельно, а затем добавить дополнительные атрибуты, необходимые производному классу. Ситуация отчасти сходна с подпрограммой clone, за исключением того, что вместо копирования существующего объекта в новый необходимо вызвать конструктор базового класса, а затем превратить новый объект базового класса в объект производного класса.

Наследование классов

Следуя установленной традиции, в системе объектов Perl наследование одного класса другим не требует особого синтаксиса. При вызове метода, для которого нет подпрограммы в пакете инвоканта, Perl исследует массив @ISA[1] этого пакета. Вот как Perl реализует наследование: каждый элемент массива @ISA данного пакета содержит имя другого пакета, в котором производится поиск отсутствующих методов. Так, в следующем примере класс Horse становится подклассом класса Critter. (Мы объявляем массив @ISA как our, потому что он должен быть переменной пакета, а не лексической переменной, объявленной через my.)

```
package Horse;
our @ISA = "Critter";
```

То же самое можно реализовать с применением прагмы parent, которая обрабатывает @ISA автоматически и загружает родительский (parent) класс:

```
package Horse;
use parent qw(Critter);
```

[1] Произносится «из э», как в «A horse is a critter» (лошадь – это животное).

Прагма parent пришла на смену устаревшей прагме base, которая выполняла те же операции, но применяла волшебство с прагмой fields, если предполагала, что поля используются суперклассами. Если вы не слышали о ней, не волнуйтесь (а просто пользуйтесь прагмой parent):

```
package Horse;
use base qw(Critter);
```

Теперь класс Horse или объект можно использовать везде, где прежде использовался класс Critter. Если ваш новый класс пройдет эту *проверку на пустой под класс*, вы будете знать, что Critter является правильным базовым классом, пригодным для наследования.

Допустим, что в $steed находится объект Horse, и вы вызываете его метод move:

```
$steed->move(10);
```

Поскольку $steed является объектом Horse, Perl попытается найти для этого метода подпрограмму Horse::move. В случае неудачи Perl не возбуждает исключительную ситуацию на этапе выполнения, а обращается к первому элементу массива @Horse::ISA, который отправляет его в пакет Critter на поиски Critter::move. Если и эта подпрограмма не будет найдена, а в Critter есть собственный массив @Critter::ISA, поиск будет продолжен в следующем пакете-предке, который мог бы предоставить метод move. Такое движение вверх по иерархии наследования будет продолжаться до достижения пакета, в котором нет @ISA.

Описанная только что ситуация называется *одиночным наследованием* (*single inheritance*): у каждого класса есть только один родитель. Такое наследование подобно связному списку родственных пакетов. Perl поддерживает также *множественное наследование* (*multiple inheritance*); достаточно добавить еще пакеты в @ISA класса. Такого рода наследование действует скорее как древовидная структура данных, потому что у каждого пакета может быть более одного непосредственного родителя. Некоторые находят, что это более сексуально.

При вызове метода *methname* с инвокантом типа *classname* Perl опробует шесть разных способов, чтобы найти подпрограмму, которую нужно использовать:

1. Сначала Perl пытается найти подпрограмму с именем *classname*::*methname* в пакете инвоканта. Если это не удается, включаются механизмы наследования, и мы переходим к шагу 2.
2. Далее Perl проверяет методы, унаследованные от базовых классов, просматривая все пакеты *parent*, которые перечислены в @*classname*::ISA, в поисках подпрограммы *parent*::*methname*. Поиск ведется слева направо, рекурсивно и сначала вглубь. Рекурсия обеспечивает проведение поиска во всех классах – прародителей, прапрародителей, прапрапрародителей и т.д.
3. Если поиск не дал результатов, Perl проверяет наличие подпрограммы с именем UNIVERSAL::*methname*.
4. В этом месте Perl отчаивается найти *methname* и начинает искать AUTOLOAD. Сначала он ищет подпрограмму с именем *classname*::AUTOLOAD.
5. В случае неудачи поиск продолжается во всех пакетах *parent*, перечисленных в @*classname*::ISA, пока не будет найдена подпрограмма *parent*::AUTOLOAD. Поиск опять ведется слева направо, рекурсивно и сначала вглубь.
6. Наконец, Perl пытается найти подпрограмму с именем UNIVERSAL::AUTOLOAD.

Perl прекращает поиск после первого же совпадения и вызывает соответствующую подпрограмму. Если подпрограмма не найдена, возбуждается исключительная ситуация (которую вы будете наблюдать нередко):

```
Can't locate object method "methname" via package "classname"
(Не могу найти метод "methname" объекта через пакет "classname")
```

Если ваша версия Perl собрана с поддержкой отладки при помощи опции компилятора -DDEBUGGING, то при запуске Perl с ключом -Do можно увидеть прохождение всех этих этапов во время поиска вызываемого метода.

По мере изложения мы будем обсуждать механизм наследования более подробно.

Наследование через @ISA

Если массив @ISA содержит больше одного имени пакета, поиск в пакетах осуществляется в порядке слева направо. Поиск ведется сначала вглубь, поэтому, если у вас есть класс Mule (мул), наследующий классы, как показано ниже:

```
package Mule;
our @ISA = ("Horse", "Donkey");
```

поиск методов, отсутствующих в Mule, сначала будет выполняться в классе Horse (и всех его предках, например, Critter) и только потом в классе Donkey (осел) и его предках.

Если отсутствующий метод найдется в базовом классе, Perl сохранит его местоположение во внутреннем кэше текущего класса для повышения эффективности, т.е. когда данный метод потребуется найти еще раз, за ним не придется далеко ходить. Изменение @ISA или определение новых методов делает кэш недействительным и заставляет Perl снова проводить поиск.

При поиске метода Perl проверяет, не создана ли циклическая иерархия наследования. Это может произойти, если два класса наследуют друг друга, в том числе косвенно, через другие классы. Попытка стать собственным прадедом парадоксальна даже для Perl, поэтому она вызывает исключение. Однако Perl не считает ошибкой наследование от нескольких классов, имеющих общих предков, что, скорее, напоминает брак между кузенами. Иерархия наследования просто перестает быть деревом и начинает выглядеть как направленный ациклический граф. Perl это не заботит, пока граф действительно остается ациклическим.

Когда устанавливается значение @ISA, присваивание происходит обычно на этапе выполнения, поэтому, если не принять мер предосторожности, код в блоках BEGIN, CHECK и INIT не будет иметь возможность использовать иерархию наследования. Одним из средств предосторожности (или удобства) является прагма parent, которая позволяет выполнять require для классов и добавлять их в @ISA на этапе компиляции. Вот как можно ее использовать:

```
package Mule;
use parent ("Horse", "Donkey"), # объявление надклассов
```

Это сокращенная форма записи следующего фрагмента:

```
package Mule;
BEGIN {
    our @ISA = ("Horse", "Donkey");
```

```
        require Horse;
        require Donkey;
    }
```

Некоторых удивляет, что включение класса в @ISA не влечет require соответствующего модуля. Это связано с тем, что система классов Perl в значительной мере ортогональна его системе модулей. В одном файле может содержаться несколько классов (поскольку они являются просто пакетами), а один пакет может упоминаться в нескольких файлах. Но в обычной ситуации, когда один пакет, один класс, один модуль и один файл в конечном итоге оказываются взаимозаменяемыми (если сильно прищуриться), прагма parent предлагает синтаксис объявления, который устанавливает наследование и загружает файлы модулей. Это одна из удобных диагоналей, о которых мы постоянно говорим.

Дополнительные сведения можно найти в описании use parent в главе 29. Прочтите также описание устаревшей прагмы base (впавшей в немилость у программистов на Perl), которая использует волшебство прагмы fields.

Альтернативный способ поиска методов

В случае множественного наследования стандартный поиск методов обходом @INC может находить неправильный метод, потому что метод более удаленного надкласса может скрыть требуемый метод, реализованный в более близком надклассе. Взгляните на дерево наследования, представленное на рис. 12.1, где класс Mule наследует два других класса, Donkey и Horse, каждый из которых наследует класс Equine. Класс Equine имеет метод color, наследуемый классом Donkey. Класс Horse имеет собственный метод color. При использовании механизма поиска по умолчанию, нельзя сказать, какой метод будет вызван, если не знать, в каком порядке просматриваются родительские классы:

```
    use parent qw(Horse Donkey); # первым будет найден Horse::Color
    use parent qw(Donkey Horse); # первым будет найден Equine::Color
```

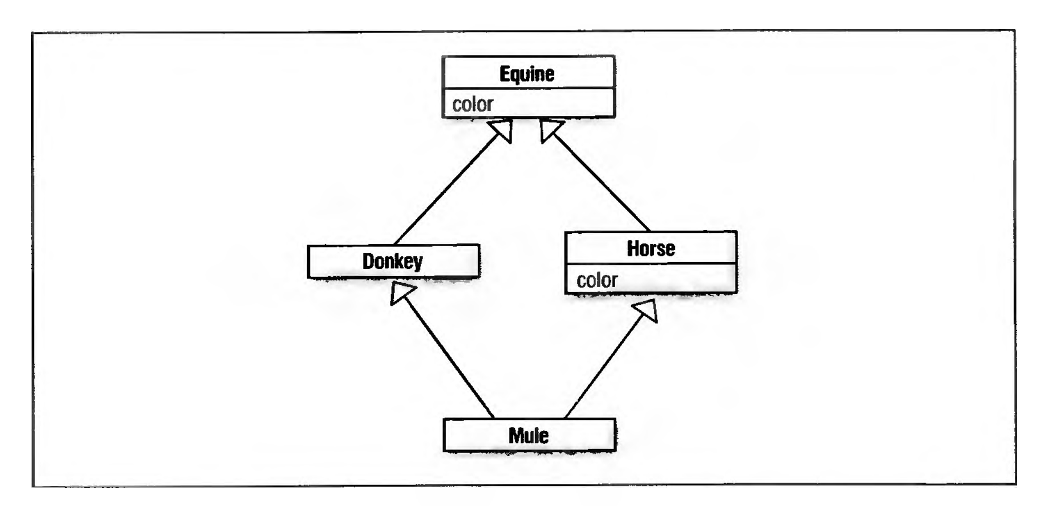


Рис. 12.1. Дерево множественного наследования

Начиная с версии v5.10 порядок обхода родительских классов можно настраивать. *Порядок поиска методов* можно установить с помощью прагмы mro (см. главу 29):

```
package Mule;
use mro 'c3';
use parent qw(Donkey Horse);
```

Алгоритм C3 так выполняет обход массивов @INC, что обнаруживает унаследованные методы в наиболее близких надклассах. Иными словами, ни один надкласс не будет просматриваться, пока поиск не будет закончен в его подклассах. Perl не станет просматривать класс Equine, пока не просмотрит класс Horse.

Если ваша версия Perl не поддерживает прагму mro, можете воспользоваться модулем MRO::Compat из архива CPAN.

Доступ к замещенным методам

Когда класс определяет метод, эта подпрограмма замещает одноименные методы во всех базовых классах. Представьте, что у вас есть объект Mule (производный от класса Horse и класса Donkey), и вы решаете вызвать метод breed (скрещивание) вашего объекта. Хотя у родительских классов есть собственные методы breed, разработчик класса Mule заместил их, снабдив класс Mule собственным методом breed. Это значит, что следующее скрещивание вряд ли будет продуктивным:

```
$stallion = Horse->new(gender => "male");
$molly = Mule->new(gender => "female");
$colt = $molly->breed($stallion);
```

Теперь допустим, что чудеса генной инженерии позволили вам решить известную проблему стерильности мула, и вы хотите перескочить через нежизнеспособный метод Mule::breed. Свой метод *можно* вызвать как обычную подпрограмму, не забыв явным образом передать инвоканта:

```
$colt = Horse::breed($molly, $stallion);
```

Однако это делается в обход наследования, что почти всегда не лучшая идея. Вполне можно представить, что подпрограммы Horse::breed не существует, потому что Horse и Donkey наследуют это свое поведение от общего родительского класса Equine. Если, с другой стороны, необходимо указать Perl, чтобы он начал поиск метода в определенном классе, выберите обычный метод вызова, но квалифицируйте имя метода именем класса:

```
$colt = $molly->Horse::breed($stallion)
```

Иногда требуется, чтобы метод производного класса действовал как обертка вокруг некоторого метода базового класса. Метод в производном классе может сам вызвать метод базового класса, выполняя дополнительные операции до или после этого вызова. Только что продемонстрированный способ записи *позволяет* также указать, в каком классе начинать поиск. Но чаще, применяя замещенные методы, желательно избавиться от необходимости явно указывать родительский класс, замещенный метод которого должен выполняться.

В таких случаях удобно использовать псевдокласс SUPER. Он позволяет вызвать замещенный метод базового класса, не указывая, в каком классе этот метод определен.[1] Следующая подпрограмма отыщет текущий пакет в @ISA, не заставляя вас указывать конкретные классы:

```
package Mule;
our @ISA = qw(Horse Donkey);
sub kick {
    my $self = shift;
    print "Мул лягается!\n";
    $self->SUPER::kick(@_);
}
```

Псевдопакет SUPER имеет значение только когда применяется *внутри* метода. Автор класса может включить SUPER в свой код, но тот, кто просто использует объекты класса, – нет.

При использовании алгоритма поиска методов C3 вместо SUPER::*METHNAME* следует использовать next::method, загружаемый прагмой use mro "c3". В отличие от SUPER, при обращении к next::method не требуется указывать имя метода, потому что оно подразумевается по умолчанию:

```
use v5.14;
package Mule;
use mro 'c3';
use parent qw(Horse Donkey);
sub kick {
    my $self = shift;
    say "Мул лягается!",
    $self->next::method(@_);
}
```

В любой точке кода на языке Perl известно, какой пакет является текущим, т.е. определен последней инструкцией package. Метод SUPER просматривает массив @ISA только того пакета, который был текущим на момент компиляции вызова SUPER. Его не интересуют ни класс инвоканта, ни пакет вызываемой подпрограммы. Это может послужить источником проблем, если попытаться определить методы в другом классе путем лишь манипуляций с именем метода:

```
package Bird;
use Dragonfly;
sub Dragonfly::divebomb { shift->SUPER::divebomb(@_) }
```

К несчастью, в этом примере будет вызван надкласс Bird, а не Dragonfly. Чтобы осуществить то, что задумывалось, нужно явно перейти в соответствующий пакет, и тогда вызов SUPER скомпилируется правильно:

```
package Bird;
use Dragonfly;
{
    package Dragonfly;
```

1 Не надо путать это с механизмом, о котором говорилось в главе 11; тот служит для замещения встроенных функций Perl, не являющихся методами объектов и потому не замещаемых путем наследования. Замещенные функции вызываются через псевдопакет CORE, а не псевдопакет SUPER.

```
    sub divebomb { shift->SUPER::divebomb(@_) }
}
```

Метод next::method страдает похожей проблемой, потому что он использует пакет, которому принадлежит вызывающий его код, чтобы определить, в каком классе следует искать. Если определить метод в классе Donkey из другого пакета, next::method не найдет его:

```
package main;
*Donkey::sound = sub { (shift)->next::method(@_) };
```

Анонимная подпрограмма фигурирует в стеке как __ANON__, поэтому next::method не сможет определить, в каком пакете она находится. Однако проблему можно решить с помощью модуля Sub::Name из архива CPAN:

```
use Sub::Name qw(subname);
*Donkey::sound =
    subname 'Donkey::sound' => sub { (shift)->next::method(@_) }
```

Как показывают эти примеры, необязательно редактировать файл модуля, чтобы добавить новые методы в существующий класс. Поскольку класс является просто пакетом, а метод – просто подпрограммой, нужно лишь определить функцию в нужном пакете, что мы здесь и сделали, и в классе внезапно появится новый метод. Никакого наследования не требуется. Значение имеет только пакет, а так как пакеты являются глобальными, то доступ к любому пакету может быть осуществлен из любого места программы. (Кстати, мы уже говорили, что собираемся на следующей неделе установить джакузи у вас в комнате?)

UNIVERSAL: первичный класс предков

Если в результате поиска в классе инвоканта и рекурсивного поиска во всех классах, являющихся его предками, не будет найдено определение метода с нужным именем, будет выполнена еще одна попытка найти метод с этим именем в особом предопределенном классе с именем UNIVERSAL. Этот пакет никогда не указывается в @ISA, но проверка в нем выполняется всегда, если поиск по @ISA не дал результатов. Можно считать UNIVERSAL всеобщим прародителем, который неявно наследуют все классы, подобным классу Object в Java или object для классов нового стиля в Python.

Ниже описаны предопределенные методы, имеющиеся в классе UNIVERSAL, а следовательно и во всех классах. Все они работают независимо от того, вызываются ли они как методы класса или как методы объекта.

INVOCANT->isa(*CLASS*)

Метод isa возвращает истинное значение, если классом ивоканта *INVOCANT* является *CLASS* или любой класс, наследующий *CLASS*. При этом *CLASS* может быть не именем пакета, а одним из встроенных типов, например, "HASH" или "ARRAY". (Однако выяснение точного типа не сулит ничего хорошего инкапсуляции и полиморфизму. Чтобы получить правильный метод, следует полагаться на диспетчеризацию методов.)

```
use IO::Handle;
if (IO::Handle->isa("Exporter")) {
    print "IO::Handle есть Exporter.\n";
}
```

```
$fh = IO::Handle->new();
if ($fh->isa("IO::Handle")) {
    print "\$fh есть какой-то объект IO.\n";
} if ($fh->isa("GLOB")) {
    print "\$fh есть ссылка на GLOB.\n";
}
```

INVOCANT->DOES(*ROLE*)

В Perl v5.10 появилось понятие *ролей*, способа включения в класс внешних методов без необходимости наследовать их, как того требует isa. Роль определяет набор поведенческих характеристик, но не указывает, как они должны поддерживаться классом. Это может быть наследование методов, их имитация, делегирование другим классам или что-то иное.

По умолчанию метод DOES действует идентично методу isa, и его можно использовать вместо isa во всех случаях. Однако, если класс делает нечто особенное, чтобы включить методы без наследования, необходимо определить метод DOES, если необходимо обеспечить получение правильного ответа.

Роли – это идея из Perl 6, и правда в том, что в Perl 5 они вообще никак не используются. Метод UNIVERSAL DOES существует, только чтобы обеспечить сотрудничество классов, которые включают методы друг друга, где метод DOES имеет большое значение. Но сам Perl не уделяет ролям никакого внимания.

INVOCANT->can(*METHOD*)

Метод can возвращает ссылку на подпрограмму, вызываемую в случае применения *METHOD* к *INVOCANT*. Если такая подпрограмма не будет найдена, возвращается undef.

```
if ($invocant->can("copy")) {
    print "Наш инвокант умеет копировать.\n";
}
```

Это можно использовать, чтобы вызывать метод только при условии его существования:

```
$obj->snarl if $obj->can("snarl");
```

При множественном наследовании это позволяет методу вызвать все замещенные методы базового класса, а не только самый левый:

```
sub snarl {
    my $self = shift;
    print "Ворчащие (snarling): @_\n";
    my %seen;
    for my $parent (@ISA) {
        if (my $code = $parent->can("snarl")) {
            $self->$code(@_) unless $seen{$code}++;
        }
    }
}
```

Мы используем хеш %seen, чтобы следить за тем, какие подпрограммы уже вызывались, и избежать повторного вызова одной и той же подпрограммы. Это могло бы произойти, будь у нескольких родительских классов общий предок.

Методы, приводящие к срабатыванию AUTOLOAD (описывается в следующем разделе), учитываются неточно, за исключением случая, когда в пакете объявлены (но не определены) подпрограммы для автоматической загрузки.

При использовании прагмы mro вместо этого метода, вероятно, лучше использовать метод next::can.

INVOCANT->VERSION(*NEED*)

Метод VERSION возвращает значение номера версии класса инвоканта *INVOCANT*, хранящееся в переменной пакета $VERSION. Если передается аргумент *NEED*, метод сравнивает его с текущей версией. Если текущая версия оказывается меньше, чем *NEED*, – возбуждает исключительную ситуацию. Этот метод использует use, чтобы определить, достаточно ли свеж модуль.

```
use Thread 1.0; # вызовет Thread->VERSION(1.0)
print "Выполняется Thread версии ", Thread->VERSION, " \n";
```

Вы можете создать собственный метод VERSION для замещения метода в UNIVERSAL. Однако при этом все классы, производные от вашего, тоже будут использовать замещающий метод. Чтобы этого не произошло, метод должен отправлять запросы версии, исходящие от других классов, снова в UNIVERSAL.

Методы в UNIVERSAL – это встроенные подпрограммы Perl, которые можно вызывать, полностью квалифицируя их и передавая два аргумента, например UNIVERSAL::isa($formobj, "HASH"). Однако при этом не выполняются некоторые проверки, поскольку $formobj может быть любым значением, а не только ссылкой. Вызов метода можно было бы заключить в ловушку eval:

```
eval { UNIVERSAL::isa($formobj, "HASH") }
```

Делать так, впрочем, не рекомендуется, поскольку ответ на ваш вопрос обычно может дать can.

```
eval { UNIVERSAL::can($formobj, $method) }
```

Если же вас беспокоит, является ли переменная $formobj объектом, и вы желаете завернуть вызов метода в eval, ее также можно использовать как объект, поскольку ответ останется прежним (вы не сможете вызвать этот метод для $formobj):

```
eval { $formobj->can( $method ) }
```

В класс UNIVERSAL можно добавлять собственные методы. (Конечно, следует проявлять осторожность: можно запутать кого-либо, кто рассчитывает на *отсутствие* метода с определенным вами именем и осуществление его автозагрузки из другого места.) Сейчас мы создадим метод copy, который могут использовать объекты всех классов, не имеющие собственного метода. При этом мы демонстрируем зрелищный сбой, если метод вызван для класса, а не объекта:

```
use Data::Dumper;
use Carp;
sub UNIVERSAL::copy {
    my $self = shift;
    if (ref $self) {
        return eval Dumper($self); # ссылки на CODE не допускаются
    } else {
```

```
        confess "UNIVERSAL::copy не может копировать класс $self";
    }
}
```

Эта стратегия Data::Dumper не работает, если объект содержит ссылки на подпрограммы, потому что они не могут быть правильно воспроизведены. Даже будь доступен исходный код, лексические привязки потерялись бы.

Автозагрузка методов

Обычно при обращении к несуществующей подпрограмме в пакете, который определяет подпрограмму AUTOLOAD, вместо возбуждения исключительной ситуации вызывается подпрограмма AUTOLOAD (см. раздел «Автозагрузка» главы 10). Для методов это действует несколько иначе. Если обычный поиск метода (через класс, его предков и, наконец, UNIVERSAL) не дает соответствия, та же последовательность поиска выполняется снова, но на этот раз производится поиск подпрограммы AUTOLOAD. Если она найдена, то вызывается в качестве метода, при этом переменная пакета $AUTOLOAD устанавливается равной полностью квалифицированному имени подпрограммы, от имени которой была вызвана AUTOLOAD.

При автозагрузке методов следует проявлять некоторую осторожность. Во-первых, подпрограмма AUTOLOAD должна немедленно осуществить возврат, если вызвана от имени метода DESTROY, если только вашей целью не была эмуляция DESTROY, имеющего в Perl особое значение, как описано в разделе «Деструкторы экземпляров» далее в этой главе.

```
sub AUTOLOAD {
    return if our $AUTOLOAD =~ /::DESTROY$/;
    ...
}
```

Во-вторых, если класс предоставляет страховку на основе AUTOLOAD, невозможно использовать UNIVERSAL::can с именем метода, чтобы проверить, безопасно ли его вызывать. Проверять AUTOLOAD нужно отдельно:

```
if ($obj->can("methname") || $obj->can("AUTOLOAD")) {
    $obj->methname();
}
```

Наконец, когда при множественном наследовании класс наследует два или более классов, в каждом из которых есть AUTOLOAD, запускается только самый левый из них, так как Perl останавливается, найдя первый AUTOLOAD.

Последние две странности легко обойти, объявив подпрограммы в пакете, чей AUTOLOAD должен управлять этими методами. Можно сделать это с помощью отдельных объявлений:

```
package Goblin;
sub kick;
sub bite;
sub scratch;
```

или с помощью прагмы subs, что более удобно, если объявляемых методов много:

```
package Goblin;
use subs qw(kick bite scratch);
```

Эти подпрограммы объявлены, но не определены, однако системе этого достаточно, чтобы считать их существующими. Они видны при проверке UNIVERSAL::can и, что более важно, на шаге 2 поиска метода, так что поиск никогда не перейдет к шагу 3, не говоря уже о шаге 4.

«Но послушайте», – воскликнете вы, – «они же вызывают AUTOLOAD, не так ли?». В конечном счете, да, но механизм этого иной. Найдя на шаге 2 заглушку метода, Perl попытается вызвать его. Если обнаружится, что метод – не то, чем ему полагается быть, снова начнется поиск метода AUTOLOAD, но на этот раз поиск начнется в классе, содержащем заглушку, что ограничивает поиск этим классом и его предками (и UNIVERSAL). Благодаря этому Perl находит для запуска правильный AUTOLOAD и умеет игнорировать методы AUTOLOAD, находящиеся в неподходящей части исходного дерева наследования.

Закрытые методы

Есть такой способ вызова метода, при котором Perl вообще игнорирует наследование. Если вместо литерального имени метода задать простую скалярную переменную, содержащую ссылку на подпрограмму, то подпрограмма вызывается непосредственно. В описании UNIVERSAL->can в предыдущем разделе последний пример вызывает все замещенные методы с помощью ссылки на подпрограмму, а не ее имени.

Интересно, что такой режим можно применять для реализации вызовов закрытых методов. Если поместить класс в модуль, лексическую область видимости файла можно будет использовать для создания закрытых методов. Сначала сохраним анонимную подпрограмму в лексической переменной с файловой областью видимости:

```
# объявление закрытого метода
my $secret_door = sub {
    my $self = shift;

};
```

Далее переменную можно использовать в этом файле, как если бы она содержала имя метода. Замыкание будет вызвано непосредственно, невзирая на наследование. Как и для любого другого метода, в качестве дополнительного аргумента передается инвокант.

```
sub knock {
    my $self = shift;
    if ($self->{knocked}++ > 5) {
        $self->$secret_door();
    }
}
```

Это позволяет собственным подпрограммам файла (методам класса) вызывать метод, недоступный вне лексической области видимости.

Деструкторы экземпляров

Как и для любого другого объекта ссылки в Perl, как только исчезает последняя ссылка на объект, его память неявно освобождается для повторного использования. Объекты позволяют перехватить управление непосредственно перед тем,

как это произойдет, путем определения подпрограммы DESTROY в пакете класса. Этот метод автоматически запускается в нужный момент, и в качестве единственного объекта ему передается объект, который вот-вот должен быть уничтожен.

Деструкторы редко требуются в Perl, потому что управление памятью осуществляется автоматически. Однако некоторые объекты могут владеть ресурсами за пределами памяти, и о них желательно было бы позаботиться; например, это касается дескрипторов файлов и соединений с базами данных.

```
package MailNotify;
sub DESTROY {
    my $self = shift;
    my $fh   = $self->{mailhandle};
    my $id   = $self->{name};
    print $fh "\n$id отключается в " . localtime() . "\n";
    close $fh, # закрыть канал обработчика почты
}
```

Для создания объекта достаточно единственного метода, даже если класс конструктора наследует один или несколько других классов. Точно так же Perl использует всего один метод DESTROY для каждого объекта, невзирая на наследование. Иными словами, Perl не станет выполнять иерархическое уничтожение. Если ваш класс замещает деструктор надкласса, вашему методу DESTROY может понадобиться вызвать метод DESTROY всех соответствующих базовых классов:

```
sub DESTROY {
    my $self = shift;
    # Проверка замещения деструктора..
    $self->SUPER::DESTROY if $self->can("SUPER::DESTROY");
    # теперь делайте свои собственные дела
}
```

Это относится только к производным классам; объект, который просто содержится в текущем объекте – как, например, одно значение в более крупном хеше – будет освобожден и уничтожен автоматически. Это одна из причин, по которым вложенность путем простого агрегирования (иногда называемая отношением «has-a» – «имеет») часто является более простой и понятной, чем наследование (отношение «is-a» – «является»). Иными словами, часто достаточно поместить один объект внутрь другого, избегая наследования, которое вносит ненужную сложность. Иногда пользователи хватаются за множественное наследование, в то время как вполне достаточно одиночного.

Явный вызов DESTROY допускается, но редко требуется. Он может даже оказаться вредным, поскольку повторный запуск деструктора одного и того же объекта может привести к неприятным последствиям.

Сборка мусора методами DESTROY

Как описывалось в разделе «Сборка мусора, циклические и слабые ссылки» главы 8, переменные, которые ссылаются сами на себя (или несколько ссылок, ссылающихся друг на друга косвенно), не освобождаются до выхода из программы (или встроенного интерпретатора). Чтобы освободить занимаемую ими память раньше, необходимо явно разорвать ссылку или ослабить ее с помощью модуля Scalar::Util из CPAN.

Другим решением может стать создание класса-контейнера, содержащего указатель на структуру данных со ссылками на саму себя. Определите метод `DESTROY` для класса, содержащего объект, который явно разрушит цикличность в структуре данных, содержащей ссылки на саму себя. Пример такого рода можно найти в главе 13 книги «Perl: библиотека программиста» Т. Кристиансена и Н. Торкингтона («Perl Cookbook») в рецепте 13.13 «Как справиться с циклическими структурами данных» (*http://my.safaribooksonline.com/0-596-00313-7/perlckbk2-CHP-11-SECT-15*).

Когда интерпретатор завершает работу, все его объекты уничтожаются, что важно для многопоточных приложений и встраиваемых приложений на языке Perl. Объекты уничтожаются в отдельном проходе, перед уничтожением обычных ссылок. Это делается, чтобы предотвратить использование методами `DESTROY` уже уничтоженных ссылок. (А также потому, что обычные ссылки утилизируются сборщиком мусора только во встроенных интерпретаторах, поскольку выход (exit) из процесса – это самый *быстрый* способ утилизации ссылок. Но при выходе не запускаются деструкторы объектов, поэтому сначала Perl делает это.)

Управление данными экземпляров

Большинство классов создает объекты, которые, по существу, являются просто структурами данных с несколькими внутренними полями (переменными экземпляра), для работы с которыми добавлены методы.

Классы Perl наследуют методы, а не данные, и пока доступ к объекту осуществляется только через вызовы методов, это отлично действует. Если требуется наследование данных, его нужно осуществлять через наследование методов. В целом, это не является необходимостью в Perl, поскольку большинство классов хранит данные своих объектов в анонимных хешах. Данные экземпляра объекта хранятся в этом хеше, служащем собственным маленьким пространством имен, которое разбирают классы, когда им нужно что-то сделать с объектом. Например, если нужно, чтобы в объекте `$city` было поле данных с именем `elevation`, можно просто обратиться к `$city->{elevation}`. Никаких предварительных объявлений не требуется. Но и у методов-оберток есть свое применение.

Допустим, мы хотим реализовать объект `Person`. Мы решили создать поле данных «name», которое, по странному совпадению, будет храниться с ключом `name` в анонимном хеше, служащем в качестве объекта. Однако нежелательно, чтобы пользователи имели прямой доступ к данным. Чтобы пользоваться преимуществами инкапсуляции, для доступа к этой переменной экземпляра пользователи должны иметь методы, не нарушающие барьер абстракции.

Например, можно создать пару методов доступа:

```
sub get_name {
    my $self = shift,
    return $self->{name};
}

sub set_name {
    my $self       = shift;
    $self->{name} = shift;
}
```

что ведет к написанию такого кода:

```
$him = Person->new();
$him->set_name("Frodo");
$him->set_name( ucfirst($him->get_name) );
```

Можно даже объединить два метода в один:

```
sub name {
    my $self = shift;
    if (@_) { $self->{name} = shift }
    return $self->{name};
}
```

Что даст, в итоге, такой код:

```
$him = Person->new();
$him->name("Frodo");
$him->name( ucfirst($him->name) );
```

Преимуществом наличия отдельных функций для каждой переменной экземпляра (для нашего класса Person это могут быть name – имя, age – возраст, height – рост и т.д.) являются непосредственность, очевидность и гибкость. Недостатком является необходимость определять в новом классе один или два почти идентичных метода для каждой переменной экземпляра. На первый взгляд, в этом нет ничего страшного, и никому не возбраняется выбрать тот способ, который больше понравится. Но если удобство более важно для вас, чем гибкость, предпочтительным может оказаться один из приемов, описанных в следующих разделах.

Обратите внимание, что мы будем изменять реализацию, а не интерфейс. Если пользователи вашего класса с уважением относятся к инкапсуляции, вы можете переходить от одной реализации к другой так, что они ничего не заметят. (Родственники из вашего дерева наследования, использующие ваш класс в качестве подкласса или надкласса, могут быть не столь покладисты, поскольку знают вас лучше, чем посторонние.) Если ваши пользователи совали свой нос в частные дела вашего класса, то неминуемо ожидающая их катастрофа – это их вина и не ваша забота. Все, что вы можете сделать, – это до конца придерживаться контракта, сохраняя интерфейс. Попытка помешать кому бы то ни было сделать что-то хоть сколько-нибудь вредное отнимет все ваше время и силы – и в конце концов все равно провалится.

Договориться с членами семьи сложнее. Если подкласс замещает метод доступа к атрибуту из надкласса, должен он обращаться к тому же полю хеша или нет? Можно привести аргументы в пользу того и другого, в зависимости от природы атрибута. В общем случае в интересах безопасности каждый метод доступа может добавлять префикс с именем своего класса к имени поля, чтобы подкласс и надкласс имели собственные варианты поля. Некоторые из приведенных ниже примеров, а также стандартный модуль Struct::Class используют именно такую стратегию защиты от подкласса. Поэтому можно встретить методы доступа вроде таких:

```
sub name {
    my $self = shift;
    my $field = __PACKAGE__ . "::name";
    if (@_) { $self->{$field} = shift }
    return $self->{$field};
}
```

В каждом из последующих примеров создается простой класс Person с полями name, race и aliases, интерфейсы которых идентичны, но реализация совершенно разная. Мы не скажем, который из них нам больше нравится, потому что они все нам нравятся, в зависимости от ситуации. О вкусах не спорят. Кто-то любит тушеного кролика, а кто-то рыбку.

Генерация методов доступа с помощью автозагрузки

Как говорилось выше, при вызове несуществующего метода у Perl есть два способа поиска метода AUTOLOAD в зависимости от того, объявлен ли метод-заглушка. Это свойство можно использовать для предоставления доступа к данным экземпляра без написания отдельной функции для каждого экземпляра. В подпрограмме AUTOLOAD имя фактически вызываемого метода можно извлечь из переменной $AUTOLOAD. Рассмотрим следующий код:

```
use Person;
$him = Person->new;
$him->name("Aragorn");
$him->race("Man");
$him->aliases( ["Strider", "Estel", "Elessar"] );
printf "%s is of the race of %s.\n", $him->name, $him->race;
print "His aliases: ", join(", ", @{$him->aliases}), ".\n";
```

Как и прежде, эта версия класса Person реализует структуру данных с тремя полями: name, race и aliases:

```
package Person;
use Carp;

my %Fields = (
    "Person::name"    => "unnamed",
    "Person::race"    => "unknown",
    "Person::aliases" => [],
);

# Следующее объявление гарантирует получение собственного автозагрузчика
use subs qw(name race aliases);

sub new {
    my $invocant = shift;
    my $class    = ref($invocant) || $invocant;
    my $self     = { %Fields, @_ }   # клонировать, как Class::Struct
    bless $self, $class;
    return $self;
}

sub AUTOLOAD {
    my $self = shift;
    # обрабатывать только методы экземпляра, но не класса
    croak "$self не является объектом" unless ref($self);
    my $name = our $AUTOLOAD;
    return if $name =~ /::DESTROY$/;
    unless (exists $self->{$name}) {
        croak "Нет доступа к полю '$name' в $self";
```

```
        }
        if (@_) { return $self->{$name} = shift }
        else    { return $self->{$name} }
    }
```

Как видите, методов с именами name, race или aliases нигде нет. Обо всем этом заботится программа AUTOLOAD. Если кто-то использует $him->name("Aragorn"), подпрограмма AUTOLOAD получит в переменной $AUTOLOAD строку с именем "Person::name". То, что имя полностью квалифицировано, приводит его как раз в тот формат, который нужен для доступа к полям объекта. Благодаря этому в случае применения данного класса в составе более развитой иерархии не возникает конфликт с такими же именами в других классах.

Генерация методов доступа с помощью замыканий

Большинство методов доступа делают, в сущности, одно и то же: они просто выбирают значение из переменной экземпляра или записывают его туда. В Perl наиболее естественный способ создания семейства функций, почти повторяющих одна другую, – определять их в цикле по замыканию. Но замыкания представляют собой анонимные функции, у которых нет имен, а методы должны быть именованными подпрограммами в таблице имен пакета класса, чтобы к ним можно было обращаться по имени. Эта проблема решается легко: нужно присвоить ссылку на замыкание переменной typeglob с нужным именем.

```
package Person;

sub new {
    my $invocant = shift;
    my $self = bless({}, ref $invocant || $invocant);
    $self->init();
    return $self;
}

sub init {
    my $self = shift;
    $self->name("unnamed")
    $self->race("unknown")
    $self->aliases([]);
}

for my $field (qw(name race aliases)) {
    my $slot = __PACKAGE__ . "::$field";
    no strict "refs";          # Так работает символическая ссылка на typeglob.
    *$slot = sub{
        my $self = shift;
        $self->{$field} = shift if @_;
        return $self->{$field};
    };
}
```

Замыкания служат самым компактным ручным способом создания группы методов доступа к данным экземпляра. Он эффективен как для компьютера, так и для программиста. Методы доступа не только совместно используют один и тот же

код (им только нужна собственная лексическая память), но и требуют минимальных изменений, если впоследствии понадобится добавить новый атрибут: нужно лишь добавить еще одно слово в список цикла `for` и, возможно, что-нибудь в метод `init`.

Использование замыканий в закрытых объектах

Приводившиеся до сих пор приемы управления данными экземпляров не предоставляли механизма «защиты» от внешнего доступа. Находясь вне класса, каждый может открыть черный ящик объекта и залезть внутрь – если он не опасается потери гарантии. Внедрение политики ограничений может привести к тому, что она станет поперек дороги тем, кто ищет пути решения стоящей перед ними задачи. Философия Perl состоит в том, что инкапсуляцию данных лучше производить с помощью надписи, которая гласит:

```
В СЛУЧАЕ ПОЖАРА
 РАЗБИТЬ СТЕКЛО
```

К такого рода инкапсуляции следует, по возможности, относиться с уважением, но, тем не менее, нужно иметь простой доступ к содержимому в аварийной ситуации – например, для отладки.

Но если вы действительно хотите обеспечить закрытость, Perl вам мешать не намерен. Perl предоставляет низкоуровневые конструкции, позволяющие окружить класс и его объекты непроницаемым щитом – щитом более прочным, чем тот, который предлагают многие популярные объектно-ориентированные языки. Ключевыми компонентами здесь служат лексические области видимости и лексические переменные в них, а замыкания играют решающую роль.

В разделе «Закрытые методы» мы разобрали, как использовать замыкания для реализации методов, которые не видны вне файла модуля. Ниже мы рассмотрим методы доступа, которые делают данные класса настолько закрытыми, что даже остальную часть класса лишают неограниченного доступа. Но все же это достаточно традиционное применение замыканий. Действительно интересный подход состоит в том, чтобы использовать замыкание в качестве самого объекта. Переменные экземпляра объекта оказываются заперты в области видимости, свободный доступ в которую имеет только сам объект, т.е. замыкание. Это очень сильная форма инкапсуляции: она не только защищает от вмешательства извне, но требует, чтобы и другие методы того же класса получали данные объекта посредством методов доступа.

Вот пример использования этой технологии. Мы применяем замыкания как для самих объектов, так и для создаваемых методов доступа:

```
package Person;
sub new {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    my $data = {
        NAME    => "unnamed",
        RACE    => "unknown",
        ALIASES => [],
    };
    my $self = sub {
```

```
        my $field = shift;
        ################################
        ### ЗДЕСЬ ПРОВЕРКА ДОСТУПА ###
        ################################
        if (@_) { $data->{$field} = shift }
        return    $data->{$field};
    };
    bless($self, $class);
    return $self;
}
# создание имен методов
for my $field (qw(name race aliases)) {
    no strict "refs"; # для доступа к таблице имен
    *$field = sub {
        my $self = shift;
        return $self->(uc $field, @_);
    };
}
```

Объект, создаваемый и возвращаемый методом new, уже не является хешем, как это было в конструкторах, рассмотренных нами ранее. Это замыкание с исключительным доступом к данным атрибутов, хранящимся в хеше, на который ссылается $data. После завершения вызова конструктора доступ к $data (и, следовательно, атрибутам) возможен только через замыкание.

При обращении вида $him->name("Bombadil") вызывающий объект, хранящийся в $self, является замыканием, обработанным bless и возвращенным конструктором. С замыканием мало что можно сделать, кроме как вызвать его; это мы и делаем с помощью $self->(uc $field, @_). Пусть вас не вводит в заблуждение стрелка: это обычное косвенное обращение к функции, а не вызов метода. Первым аргументом является строка name, а оставшиеся аргументы представляют все остальное, что мы передали.[1] Когда выполнение переходит внутрь замыкания, ссылка на хеш внутри $data снова становится доступной. После этого замыкание может разрешить доступ или отказать в нем по своему усмотрению.

Никто за пределами объекта замыкания не имеет непосредственного доступа к этим сильно закрытым данным экземпляра, даже другие методы класса. Они могли бы попытаться обратиться к замыканию таким же способом, как это делают методы, генерируемые в цикле for, например, присваивая значение переменной экземпляра, о которой классу ничего не известно. Но такой подход легко блокируется в участках кода конструктора, где можно найти комментарии по поводу проверки доступа. Прежде всего, нужна общая преамбула:

```
use Carp;
local $Carp::CarpLevel = 1; # Делает сообщения croak короткими
my ($cpack, $cfile) = caller();
```

Теперь рассмотрим каждую из проверок. Первая проверяет существование указанного имени атрибута:

```
croak "Недопустимое поле $field в объекте"
    unless exists $data->{$field};
```

---

[1] **Конечно, двойное обращение к функциям выполняется медленно, но если вам важна скорость, станете ли вы вообще использовать объекты?**

Следующая проверка разрешает вызов только из того же файла:

```
carp "Непосредственный доступ из сторонних файлов запрещен"
    unless $cfile eq __FILE__;
```

Следующая проверка разрешает вызов только из того же пакета:

```
carp "Непосредственный доступ запрещен стороннему пакету ${cpack}:
    unless $cpack eq __PACKAGE__;
```

А эта проверка разрешает вызов только из классов, наследующих наш:

```
carp "Непосредственный доступ запрещен недружественному классу ${cpack}::"
    unless $cpack->isa(__PACKAGE__);
```

Все эти проверки блокируют только доступ без посредника. Пользователи класса, которые вежливо применяют методы доступа, таких ограничений не имеют. Perl дает вам возможность проявлять свою привередливость в любой мере. К счастью, привередничать мало кто хочет.

Но иногда приходится быть разборчивыми. Разборчивость нужна тем, кто пишет программное обеспечение для управления полетами. Если вы хотите или обязаны быть одним из них и предпочитаете использовать готовый код, а не изобретать все самостоятельно, возьмите из CPAN модуль `Tie::SecureHash` Дамиана Конвея (Damian Conway). В нем реализованы хеши с ограниченным доступом, поддерживающие разборчивость на открытом, защищенном и закрытом уровнях. В нем решаются также проблемы наследования, которые мы проигнорировали в предыдущем примере. Дамиан создал также еще более амбициозный модуль `Class::Contract`, который накладывает на гибкую систему объектов Perl формальный режим разработки программного обеспечения. Список функций этого модуля выглядит, как перечень из учебника по разработке программного обеспечения, написанного профессором информатики,[1] включая принудительную инкапсуляцию, статическое наследование и проверку условия проектирования по контракту (design-by-contract condition) для объектно-ориентированного Perl наряду с декларативным синтаксисом определения атрибутов, методов, конструкторов и деструкторов на уровне объекта и класса, а также предусловиями, постусловиями и инвариантами классов. Уф!

Новые приемы

Начиная с версии v5.6, в Perl появилась возможность объявлять методы, возвращающие l-значение. Это делается с помощью атрибута `lvalue` подпрограмм (не путайте с атрибутами объектов). Эта экспериментальная возможность позволяет обращаться с методом как с чем-то, что может появиться слева от знака равенства:

```
package Critter;

sub new {
    my $class = shift;
    my $self = { pups => 0, @_ }  # Замещение значения по умолчанию.
```

---

[1] Попробуйте угадать, кем работает Дамиан. Между прочим, весьма рекомендуем его книгу «Object Oriented Perl» (Manning).

```
    bless $self, $class;
}

sub pups : lvalue {                # Присваивание pups() будет выполнено позже.
    my $self = shift;
    $self->{pups};
}

package main;
$varmint = Critter->new(pups => 4),
$varmint->pups *= 2;               # Присваивание $varmint->pups!
$varmint->pups =~ s/(.)/$1$1/;     # Изменение $varmint->pups по месту!
print $varmint->pups;              # Теперь у нас 88 pups.
```

Это позволяет обращаться с методом `$varmint->pups` как с переменной, в то же время придерживаясь инкапсуляции. См. раздел «Атрибут lvalue» главы 7.

Если вы работаете с версией Perl, поддерживающей многопоточную модель выполнения, и хотите, чтобы только один поток мог вызывать некоторый метод с объектом, используйте для этого атрибуты `locked` и `method`:

```
sub pups : locked method {
    ...
}
```

Когда поток выполнения попытается вызвать метод `pups` объекта, перед его выполнением Perl заблокирует объект, предотвращая возможность одновременного вызова из других потоков. См. раздел «Атрибуты методов» главы 7.

Управление данными класса

Мы рассмотрели несколько способов организации доступа к данным в объектах. Однако иногда нужно иметь некоторое состояние, общее для всех объектов класса. Такие переменные являются глобальными для всего класса, а не атрибутами лишь одного его экземпляра, и не зависят от того, через какой экземпляр класса (объект) осуществляется к ним доступ. (Программисты на C++ могут представить их как статические члены-данные.) Вот некоторые ситуации, в которых может оказаться удобным использование переменных класса:

- Ведение учета всех когда-либо созданных или действующих в данный момент объектов.
- Ведение списка всех объектов, которые можно обойти в цикле.
- Хранение имени дескриптора файла журнала, используемого методом отладки всего класса.
- Ведение учета общих данных, например общей суммы денег, выданных всеми банкоматами в сети за данные сутки.
- Отслеживание последнего созданного классом объекта или объекта, к которому чаще всего осуществляется доступ.
- Поддержка кэша хранимых в памяти объектов, воссозданных из постоянной памяти.
- Поддержка таблицы обратного поиска объектов по значению атрибута.

Вопрос сводится к принятию решения о том, где хранить общие атрибуты. В Perl нет специального синтаксического механизма для объявления как атрибутов класса, так и атрибутов экземпляра. Но Perl предоставляет разработчику набор мощных, но гибких функций, которые можно уникальным образом использовать в зависимости от конкретных условий. В результате можно выбрать механизм, наиболее подходящий в данной ситуации, а не полагаться на чужие архитектурные решения. С другой стороны, можно обратиться к архитектурным решениям, которые кто-то другой упаковал и положил в CPAN. Как всегда, TMTOWTDI.

Как и во всем, что относится к классам, следует избегать непосредственного обращения к данным класса, особенно из-за пределов самого класса. Нельзя обеспечить инкапсуляцию, если после тщательной подготовки ограничивающих методов доступа к переменным экземпляра позволить кому угодно укокошить переменные класса непосредственно, например, выполнив `$SomeClass::Debug = 1`. Чтобы установить четкую преграду между интерфейсом и реализацией, можно создать методы доступа к данным класса, аналогичные тем, которые применяются для доступа к данным экземпляра.

Представим, что нужно вести глобальный учет поголовья объектов `Critter`. Мы будем хранить это число в переменной пакета, но предоставим метод с именем `population`, благодаря которому пользователям класса не потребуется знать ничего о механизме реализации.

```
Critter->population()  # Доступ через имя класса
$gollum->population()  # Доступ через экземпляр
```

Поскольку класс в Perl является просто пакетом, естественнее всего хранить данные класса в переменной пакета. Ниже приведена простая реализация такого класса. Метод `population` игнорирует переданный инвокант и возвращает текущее значение переменной пакета, `$Population`. (Некоторые программисты предпочитают именовать глобальные переменные с заглавной буквы.)

```
package Critter;
our $Population = 0;
sub population { return $Population }
sub DESTROY { $Population-- }
sub spawn {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    $Population++;
    return bless { name => shift || "anon" }, $class;
}
sub name {
    my $self = shift;
    $self->{name} = shift if @_;
    return $self->{name};
}
```

Чтобы методы данных класса работали как методы доступа к данным, следует делать так:

```
our $Debugging = 0;     # начальные данные класса
sub debug {
    shift;              # намеренное игнорирование инвоканта
```

```
        $Debugging = shift if @_;
        return $Debugging;
    }
```

Теперь можно установить общий уровень отладки в классе или любом из его экземпляров.

Являясь переменной пакета, `$Debugging` доступна глобально. Но если изменить `our` на `my`, то ее сможет видеть только последующий код в том же файле. Можно пойти еще дальше: отобрать неограниченный доступ к атрибутам класса даже у остальной части самого класса. Поместите объявление переменной в область видимости блока:

```
    {
        my $Debugging = 0;     # данные класса с лексической областью видимости
        sub debug {
            shift;             # намеренное игнорирование инвоканта
            $Debugging = shift if @_;
            return $Debugging;
        }
    }
```

Теперь никто не сможет читать или записывать атрибуты класса без использования метода доступа, так как только его подпрограмма находится в той же области видимости, что и переменная, и имеет к ней доступ.

В производном классе, унаследовавшем методы доступа к данным класса, они по-прежнему будут обращаться к исходным данным независимо от того, определялись ли переменные с помощью `our` или `my`. Данные не зависят от пакета. Можно рассматривать это как выполнение методов в классе, где они были определены, а не в классе, вызвавшем их.

Для одних типов классов такой подход работает, а для других – нет. Допустим, мы создаем подкласс `Warg`[1] класса `Critter`. Если потребуется вести отдельный учет поголовья варгов, класс `Warg` не должен будет наследовать метод `population` класса `Critter`, поскольку этот метод, как мы его написали, всегда возвращает значение `$Critter::Population`.

Необходимо решать в каждом конкретном случае отдельно, имеет ли смысл атрибутам класса зависеть от пакета. Если нужны атрибуты, зависящие от пакета, используйте класс инвоканта для поиска пакета с данными класса:

```
    sub debug {
        my $invocant = shift;
        my $class    = ref($invocant) || $invocant;
        my $varname  = $class . "::Debugging";
        no strict "refs";      # для доступа к данным пакета через таблицу имен
        $$varname = shift if @_;
        return $$varname;
    }
```

Мы временно отменяем ограничение на ссылки, потому что иначе не смогли бы использовать полностью квалифицированное символическое имя для глобальной переменной пакета. Это совершенно обоснованно: поскольку, по определению, все

---

[1] Варг, мифологическое существо – огромный волк. – *Прим. перев.*

переменные пакета располагаются в пакете, нет ничего плохого в обращении к ним через таблицу имен этого пакета.

Другой подход состоит в том, чтобы сделать все данные, необходимые объекту, – даже глобальные данные класса – доступными через этот объект (или передавать в качестве параметров). С этой целью часто приходится создавать специальный конструктор для каждого класса или, по крайней мере, специальную процедуру инициализации, вызываемую конструктором. Конструктор или процедура инициализации записывают ссылки на любые данные класса непосредственно в сам объект, поэтому в будущем их не придется искать. Методы доступа берут ссылки на данные из объекта.

Каждый метод не ведет сложный поиск данных класса; вместо этого объект сообщает методу, где эти данные находятся. Этот подход хорошо работает, только если методы доступа к данным класса вызываются как методы экземпляра, потому что данные класса могут храниться в недоступных лексических переменных, до которых нельзя добраться с помощью имени пакета.

Как ни крути, а данные класса, зависящие от пакета, всегда несколько опасны. Значительно надежнее, если при наследовании метода доступа к данным класса наследуются и данные состояния, с которыми он работает. Многочисленные и более проработанные подходы к управлению данными класса можно найти на странице руководства *perltoot*. Однако вам придется поблуждать в них.

Лось в посудной лавке (Moose)

Выше рассказывалось о встроенной объектной системе Perl, но существует другая объектная система, которая нравится программистам на Perl. Применяя приемы метаобъектного программирования, модуль Moose предоставляет множество интересных возможностей. Модуль Moose обладает намного более широкими возможностями, чем можно охватить в этой книге (и в действительности заслуживает отдельной книги), но чтобы дать некоторое представление:

```
use v5.14;

package Stables 1.01 {
    use Moose;

    has "animals" => (
        traits  => ["Array"],
        is      => "rw",
        isa     => "ArrayRef[Animal]",
        default => sub { [] },
        handles => {
            add_animal  => "push",
            add_animals => "push",
        }
    );

    sub roll_call {
        my($self) = @_;

        for my $animal ($self->animals) {
            say "Some ", $animal->type,
```

```
                " named ", $animal->name,
                " is in the stable";
            }
        }
    }

    package Animal 1.01 {
        use Moose;

        has "name" => (
            is       => "rw",
            isa      => "Str",
            required => 1,
        );

        has "type" => (
            is      => "rw",
            isa     => "Str",
            default => "animal",
        );
    }

    my $stables = Stables->new;

    $stables->add_animal(
        Animal->new(name => "Mr. Ed", type => "horse")
    );

    $stables->add_animals(
        Animal->new(name => "Donkey",   type => "donkey"),
        Animal->new(name => "Lampwick", type => "donkey"),
        Animal->new(name => "Trigger",  type => "horse" ),
    );

    $stables->roll_call;
```

Модуль Moose может существенно упростить жизнь разработчику классов. Модуль Moose из пакета Stables предоставляет возможности, которые утомительно было бы реализовывать вручную. Вызов has определяет методы доступа с указанными свойствами.

конструктор по умолчанию с аргументами по умолчанию

В пакетах Stables и Animal нет явного конструктора. Модуль Moose заботится обо всем этом автоматически. Если потребуется что-то особенное, всегда можно определить свой конструктор. Атрибут name в классе Animal является обязательным, а атрибут type имеет значение по умолчанию.

проверка параметров

В пакете Stables в строке has animals тип значения объявляется как массив ссылок ArrayRef, содержащий объекты Animal. Ключ default определяет, что делать, если конструктор вызван без аргументов (поскольку required равно 0). Модуль Moose проверит, является ли значение, переданное методу add_animals, объектом Animal.

характеристики (traits)

Ключ `traits` **определяет поведение метода доступа. Поскольку его значением является ссылка на массив, вам, вероятно, понадобится применять к нему операции, используемые при работе с массивами. Ссылка на хеш** `handles` **отображает имена, которые вы сможете использовать как имена методов. Методы** `add_animal` **и** `add_animals` **соответствуют методу** `push` **массива** `Array`**.**

Это всего лишь простой пример. Модуль `Moose` **обладает намного более широкими возможностями. Поближе познакомиться с ним можно на веб-сайте *http://moose.perl.org*.**

Существуют и другие модули, реализующие подобные интерфейсы. Каркас `Mouse` **представляет собой урезанную версию** `Moose` **и призван смягчить проблемы производительности за счет исключения некоторых редкоиспользуемых особенностей.** `Moo` **– еще одна урезанная версия** `Moose`**, без поддержки XS-реквизитов, упрощающая развертывание. А каркас** `Mo` **еще меньше.**

Резюме

Вот и все, что можно тут рассказать, за исключением всего прочего. Вам остается лишь пойти и купить книгу по методам объектно-ориентированного проектирования и биться об нее лбом ближайшие полгода или около того.

13

Перегрузка

Объекты, конечно, хороши, но иногда они *слишком* хороши. Иногда хочется, чтобы их поведение больше походило на поведение обычных типов данных. Но на этом пути есть проблема: объекты представляются ссылками, а возможности работы со ссылками невелики: ссылки нельзя складывать, выводить или (с какой-нибудь пользой) применять к ним многие встроенные операторы Perl. По сути, их можно только разыменовывать. Вот и приходится то и дело писать вызовы методов, например:

```
print $object->as_string;
$new_object = $subject->add($object);
```

Такое явное разыменование, вообще говоря, дело полезное, так как при этом не спутаешь ссылки с их объектами, кроме как в случаях, когда путаница создается намеренно. Один из таких случаев мы сейчас и рассмотрим. Если при разработке класса использовался прием *перегрузки* (*overloading*), можно сделать вид, что никаких ссылок нет, и просто сказать:

```
print $object;
$new_object = $subject + $object;
```

При перегрузке встроенного оператора Perl вы определяете, как он должен себя вести, будучи применен к объектам определенного класса. Перегрузка применяется в ряде стандартных модулей Perl, например `Math::BigInt`. Последний позволяет создавать объекты `Math::BigInt`, ведущие себя как обычные целые числа, но имеющие неограниченный размер. Их можно складывать оператором +, делить оператором /, сравнивать оператором <=> и выводить функцией `print`.

Обратите внимание, что перегрузка (overloading) – это совсем не то же самое, что автозагрузка (autoloading), которая означает загрузку отсутствующих функций или методов по требованию. И перегрузка – совсем не то же самое, что замещение (overriding), при котором одна функция или метод маскируют другую функцию. Перегрузка ничего не скрывает; благодаря ей бессмысленная до того операция над ссылкой приобретает смысл.

Прагма overload

Прагма overload **осуществляет перегрузку операторов. Ей передается список пар ключ/значение, состоящий из операторов и связанных с ними действий:**

```
package MyClass;

use overload  "+" => \&myadd,              # ссылка на код
             "<" => "less_than",          # именованный метод
           "abs" => sub { return @_ };  # анонимная подпрограмма
```

Если теперь попытаться сложить два объекта MyClass, **для формирования результата будет вызвана подпрограмма** myadd.

Если попытаться сравнить два объекта MyClass **с помощью оператора <, Perl обнаружит, что нужное действие определяется строкой и будет интерпретировать эту строку как имя метода, а не просто подпрограммы. В приведенном примере метод** less_than **может быть предоставлен самим пакетом** MyClass **или унаследован классом** MyClass **от базового класса, а вот подпрограмма** myadd **должна быть предоставлена текущим пакетом. Анонимная подпрограмма для** abs **предоставляется еще более непосредственным способом. Каким бы образом ни предоставлялись эти программы, будем называть их *обработчиками* (*handlers*).**

Для унарных операторов (т.е. принимающих единственный аргумент, как abs**) указанный для класса обработчик вызывается всякий раз, когда оператор применяется к объекту этого класса.**

Для бинарных операторов, таких как + или <, обработчик вызывается, если первый операнд является объектом класса *или* если второй операнд является объектом класса, а для первого операнда перегрузка не определена. Поэтому можно сказать:

```
$object + 6
```

или:

```
6 + $object
```

не беспокоясь о порядке операндов. (Во втором случае операнды при передаче обработчику *поменяются местами*.) Для выражения

```
$animal + $vegetable
```

$animal **(животное) и** $vegetable **(овощ) являются объектами разных классов, в каждом из которых определен перегруженный оператор +, и для выполнения операции будет использована перегрузка** $animal. **(Будем надеяться, что животному понравятся овощи.)**

В Perl есть только один тернарный («тринарный») оператор, ?:, **и перегрузить его нельзя. К счастью.**

Обработчики перегрузки

При выполнении перегруженного оператора соответствующему обработчику передается три аргумента. Первые два – операнды. Если оператор использует только один аргумент, то вторым аргументом является undef.

Третий аргумент указывает – допускается ли перестановка первых двух аргументов. Даже по правилам обычной арифметики некоторые операции, например сложение и умножение, безразличны к порядку своих аргументов, а другие, например вычитание и деление, нет.[1] Взгляните на разницу между:

```
$object - 6
```

и:

```
6 - $object
```

Если первые два аргумента обработчика могут меняться местами, то третий имеет истинное значение. Иначе третий аргумент будет иметь ложное значение, и в этом случае есть также более тонкое различие: если обработчик вызван другим обработчиком оператора с присваиванием (как для +=, где положение знака + определяет порядок сложения), то третий аргумент не просто имеет ложное значение, но undef. Это различие позволяет выполнить некоторую оптимизацию.

В качестве примера приведем класс, позволяющий работать с ограниченным диапазоном чисел. В нем + и - перегружаются так, что результат сложения или вычитания ограничивается, чтобы попасть в диапазон от 0 до 255:

```
package ClipByte;

    use overload "+" => \&clip_add,
                 "-" => \&clip_sub;

    sub new {
        my $class = shift;
        my $value = shift;
        return bless \$value => $class;
    }

    sub clip_add {
        my ($x, $y) = @_;
        my ($value) = ref($x) ? $$x : $x;
        $value     += ref($y) ? $$y : $y;
        $value = 255 if $value > 255;
        $value =   0 if $value < 0;
        return bless \$value => ref($x);
    }

    sub clip_sub {
        my ($x, $y, $swap) = @_;
        my ($value) = (ref $x) ? $$x : $x;
        $value     -= (ref $y) ? $$y : $y;
        if ($swap) { $value = -$value }
        $value = 255 if $value > 255;
        $value =   0 if $value < 0;
        return bless \$value => ref($x);
    }
```

[1] Перегружаемые объекты не обязаны придерживаться правил обычной арифметики, но вообще лучше не вызывать у людей удивление. Странно, но многие языки совершают ошибку, перегружая + операцией конкатенации строк, что не только не коммутативно, но даже аддитивно с натяжкой. В Perl вы найдете другой подход.

```
package main;

    $byte1 = ClipByte->new(200);
    $byte2 = ClipByte->new(100);

    $byte3 = $byte1 + $byte2; # 255
    $byte4 = $byte1 - $byte2; # 100
    $byte5 = 150 - $byte2:    # 50
```

Обратите внимание, что все функции здесь по необходимости являются конструкторами, поэтому каждая из них использует bless **для связывания ссылки на свой новый объект с текущим классом, чем бы он ни был; мы предполагаем, что наш новый класс может наследоваться. Мы также предполагаем, что если** $y **является ссылкой, то это ссылка на объект созданного нами типа. Вместо проверки** ref($y) **мы могли бы вызывать** $y->isa("ClipByte"), **чтобы быть абсолютно точными (но эта проверка выполнялась бы медленнее).**

Перегружаемые операторы

Перегружать можно только определенные операторы (табл. 13.1), а также в хеше %overload::ops, **который становится доступен в области действия директивы** use overload, **хотя разбивка по категориям в нем несколько иная.**

Таблица 13.1. Перегружаемые операторы

Категория	Операторы
Преобразование	"" 0+ bool qr
Арифметические	+ - * / % ** x neg
Логические	!
Поразрядные	& | - ^ ! << >>
Присваивания	+= -= *= /= %= **= x= = <<= >>= ++ --
Сравнения	== < <= > >= != <=> lt le gt ge eq ne cmp
Математические	atan2 cos sin exp abs log sqrt int
Итеративные	<>
Проверка файлов	-X
Разыменования	${} @{} %{} &{} *{}
Сопоставления	~~
Псевдо	nomethod fallback =

Обратите внимание, что neg, bool, nomethod **и** fallback **фактически не являются операторами Perl. Пять разыменовывающих операторов,** qr, "" **и** 0+ **тоже *не похожи* на операторы. Тем не менее, все они могут выступать как ключи в списке параметров директивы** use overload. **На самом деле, это не представляет проблемы. Откроем маленький секрет: мы слегка мошенничаем, говоря, что прагма** overload **перегружает операторы. Она перегружает производимые операции, когда вызывается явно через «официальные» операторы или неявно через какой-либо связанный оператор. (Упомянутые псевдооператоры могут вызываться только неявно.) Иными словами, перегрузка производится не на синтаксическом уровне, а на семантическом.**

Цель не в том, чтобы хорошо выглядеть, а в том, чтобы делать то, что нужно. Обобщить можете сами.

Заметьте также, что = *не* перегружает оператор присваивания, как можно было бы предположить. Это было бы неправильно. Подробнее об этом ниже.

Начнем с обсуждения операторов преобразования – не потому, что они самые очевидные (это не так), а потому что они самые полезные. Многие классы перегружают только преобразование в строку, задаваемое ключом "", и больше ничего. (Да, это действительно две двойных кавычки подряд.)

Операторы преобразования: "", 0+, bool, qr.

Первые три ключа позволяют задать режимы автоматического преобразования в строки, числа и логические значения соответственно.

Четвертый ключ используется, когда объект интерполируется или используется как регулярное выражение, включая ситуации, когда объект играет роль правого операнда операторов =~ и !~. Подпрограмма qr должна возвращать скомпилированное регулярное выражение или ссылку на скомпилированное регулярное выражение, как делает настоящая подпрограмма qr, а любые дальнейшие перегрузки в возвращаемом значении будут игнорироваться.

Мы говорим, что происходит *преобразование в строку* (*stringification*), если нестроковая переменная используется в качестве строки. Это происходит, когда переменная преобразуется в строку при выводе, интерполяции, конкатенации и даже при использовании в качестве ключа хеша. Преобразование также служит причиной появления чего-то вроде SCALAR(0xba5fe0) при попытке вывести объект.

Мы говорим, что происходит *преобразование в число* (*numification*), когда нечисловая переменная преобразуется в число в каком-либо числовом контексте, например в математическом выражении, индексе массива или даже операторе диапазона (..).

Наконец, хотя никто не посмеет назвать это *boolification*, но можно определить способ интерпретации объекта в логическом контексте (таком, как if, unless, while, for, and, or, &&, ||, ?: или блок выражения grep) путем создания логического обработчика.

Любые из этих трех операторов преобразования могут *самогенерироваться*, если имеется один из них (что такое самогенерация, мы разъясним позже). Обработчики могут возвращать любые значения. Отметьте, что если операция, запустившая преобразование, тоже перегружена, *ее* перегрузка происходит сразу после преобразования.

Вот пример перегруженного оператора "", который запускает обработчик as_string объекта при преобразовании в строку. Не забывайте заключать в кавычки сами кавычки:

```
package Person;

use overload q("") => \&as_string;

sub new {
    my $class = shift;
    return bless { @_ } => $class;
}
```

```
sub as_string {
    my $self = shift;
    my ($key, $value, $result);
    while (($key, $value) = each %$self) {
        $result .= "$key => $value\n";
    }
    return $result;
}

$obj = Person->new(height => 72, weight => 165, eyes => "темно-карие");

print $obj;
```

Вместо чего-то вроде Person=HASH(0xba1350), этот код выведет (в порядке хеша):

```
weight => 165
height => 72
eyes => темно-карие
```

(Искренне надеемся, что вес и рост этого человека выражаются не в килограммах и сантиметрах.)

Арифметические операторы: +, -, *, /, %, **, x, ., neg

Все они должны быть вам знакомы, за исключением neg, который является специальным ключом перегрузки унарного минуса: знака - в -123. Различие между ключами neg и - позволяет назначить различное поведение унарному минусу и бинарному минусу, обычно называемому вычитанием.

Если вы перегружаете -, но не neg, а затем пытаетесь использовать унарный минус, Perl эмулирует для вас обработчик neg. Это называется ***самогенерацией*** (***autogeneration***); при этом некоторые операторы можно разумным образом вывести из других (в допущении, что перегруженные операторы будут иметь такие же связи, как у обычных операторов.) Так как унарный минус можно выразить как функцию бинарного минуса (т.е. -123 эквивалентно 0 - 123), Perl не заставляет перегружать neg, поскольку можно обойтись -. (Конечно, если вы определили, что бинарный минус делит второй аргумент на первый, унарный минус станет отличным способом возбуждения исключительной ситуации деления на ноль.)

Конкатенация через оператор . может самогенерироваться через обработчик преобразования в строку (смотри описание "" выше).

Логический оператор: !

Если обработчик для ! не задан, он может самогенерироваться с помощью обработчиков bool, "" или 0+. Если вы перегружаете оператор !, оператор not тоже будет демонстрировать заданное вами поведение. (Помните наш маленький секрет?)

Вас может удивить отсутствие других логических операторов, но большинство логических операторов не могут перегружаться, потому что они вычисляются по короткой схеме. Являясь, в действительности, операторами управления порядком выполнения инструкций, они должны иметь возможность откладывать вычисление некоторых своих аргументов. Это также служит причиной того, что оператор ?: не перегружается.

Поразрядные операторы: &, |, ~, ^, <<, >>

Оператор ~ является унарным, а все остальные – бинарными. Вот как можно перегрузить >> для реализации чего-то вроде chop:

```
package ShiftString;

use overload
    ">>"   => \&right_shift,
    q("")  => sub { ${ $_[0] } };

sub new {
    my $class = shift,
    my $value = shift;
    return bless \$value => $class;
}

sub right_shift {
    my ($x, $y) = @_;
    my $value = $$x;
    substr($value, -$y) = "";
    return bless \$value => ref($x);
}

$camel = ShiftString->new("Camel");
$ram = $camel >> 2;
print $ram;              # Cam
```

Операторы присваивания: +=, -=, *=, /=, %=, **=, x=, .=, <<=, >>=, ++, --

Эти операторы присваивания могут изменять значения своих аргументов или оставлять их как есть. Результат присваивается левому операнду, только если новое значение отличается от старого. Благодаря этому один и тот же обработчик может использоваться для перегрузки как +=, так и +. Это допустимо, но редко рекомендуется, поскольку согласно семантике, описываемой далее в разделе «Когда отсутствует обработчик перегрузки (nomethod и fallback)», Perl в любом случае вызовет обработчик для +, предполагая что += не перегружен непосредственно.

Конкатенация (.=) может самогенерироваться с использованием преобразования в строку, за которым следует обычная конкатенация строк. Операторы ++ и -- могут самогенерироваться из + и - (или += и -=).

Предполагается, что аргументы в обработчиках ++ и -- *мутируют* (изменяются). В обработчике можно сделать так, чтобы автодекрементирование работало как с буквами, так и с числами:

```
package MagicDec;

use overload
    q(--) => \&decrement,
    q("") => sub { ${ $_[0] } };

sub new {
    my $class = shift;
    my $value = shift;
    bless \$value => $class;
}
```

```
        sub decrement {
            my @string = reverse split(//, ${ $_[0] } )
            my $i;
            for ($i = 0; $i < @string; $i++ ) {
                last unless $string[$i] =~ /a/i;
                $string[$i] = chr( ord($string[$i]) + 25 );
            }
            $string[$i] = chr( ord($string[$i]) - 1 );
            my $result = join("" => reverse @string);
            $_[0] = bless \$result => ref($_[0]);
        }

    package main;

    for $normal (qw/perl NZ Pa/) {
        $magic = MagicDec->new($normal);
        $magic--;
        print "$normal стал $magic\n";
    }
```

Этот код выведет:

```
perl стал perk
NZ стал NY
Pa стал Oz
```

Это точная противоположность встроенного в Perl волшебного оператора автоинкрементирования строки.

Операция ++$a может самогенерироваться с использованием $a += 1 или $a = $a + 1, а $a-- – с использованием $a -= 1 или $a = $a - 1. Однако при этом нет поведения копирования, которое демонстрирует настоящий оператор ++. См. раздел «Конструктор копирования» далее в этой главе.

Операторы сравнения: ==, <, <=, >, >=, !=, <=>, lt, le, gt, ge, eq, ne, cmp

Если оператор <=> перегружен, он может применяться для самогенерации поведения операторов <, <=, >, >=, == и !=. Аналогично, если перегружается cmp, он может использоваться для самогенерации поведения lt, le, gt, ge, eq и ne.

Обратите внимание, что перегрузка cmp не позволяет так легко, как хотелось бы, сортировать объекты, поскольку сравниваться будут не сами объекты, а их строковые версии. Чтобы это сделать, потребуется также перегрузить "".

Математические функции: atan2, cos, sin, exp, abs, log, sqrt, int

Если функция abs недоступна, она может самогенерироваться из < или <=> в сочетании с унарным минусом или вычитанием.

Перегруженный - (минус) может применяться для самогенерации отсутствующих обработчиков унарного минуса или функции abs, которая может перегружаться отдельно. (Да, мы знаем, что abs напоминает функцию, а унарный минус напоминает оператор, но в Perl они не настолько различаются.)

Исторически сложилось так, что функция int выполняет округление в сторону 0 (см. описание int в главе 27), поэтому для объектов, действующих подобно вещественным типам, она должна делать то же самое, чтобы не вызывать недоумение пользователей таких объектов.

Итеративный оператор: `<>`

Обработчик `<>` может запускаться путем использования `readline` (при чтении из дескриптора файла, как в `while (<FH>)`) или `glob` (при поиске файлов по шаблону, как в `@files = <* *>`).

```
package LuckyDraw,

use overload
    "<>" => sub {
                my $self = shift;
                return splice @$self, rand @$self, 1;
            };

sub new {
    my $class = shift,
    return bless [@_] => $class;
}

package main;

$lotto = new LuckyDraw 1    51;

for (qw(1-й 2-й 3-й 4-й 5-й 6-й)) {
    $lucky_number = <$lotto>;
    print "$_ счастливый номер: $lucky_number.\n";
}

$lucky_number = <$lotto>;
print "\nИ дополнительный номер: $lucky_number.\n";
```

В Калифорнии[1] этот код выведет:

```
1-й счастливый номер: 18
2-й счастливый номер: 11
3-й счастливый номер: 40
4-й счастливый номер: 7
5-й счастливый номер: 51
6-й счастливый номер: 33

И дополнительный номер: 5
```

Операторы проверки файлов

Ключ `-X` используется для определения подпрограммы, обрабатывающей все операторы проверки файлов, такие как `-f`, `-x` и т.д. См. табл. 3.4 в разделе «Именованные унарные операторы и операторы проверки файлов» главы 3.

Нельзя выполнить перегрузку операторов проверки файлов выборочно. Чтобы различать их, во втором аргументе передается буква, следующая после знака «-» (т.е. в позиции, где бинарные операторы получают второй операнд).

Вызов перегруженного оператора проверки файлов не влияет на значение `stat`, связанное со специальным дескриптором файла `_`. Оно по-прежнему будет ссылаться на результат последнего вызова `stat`, `lstat` или выполнения неперегруженного оператора проверки файлов.

---

1 Речь идет о лотерее CALIFORNIA SUPERLOTTO. – *Прим. ред.*

Перегрузка этих операторов стала возможной, начиная с версии v5.12.

Операторы разыменования: ${}, @{}, %{}, &{}, *{}

Попытки разыменования ссылок на скаляры, массивы, хеши, подпрограммы и glob могут перехватываться путем перегрузки этих пяти символов.

В электронной документации по директиве overload показано, как с помощью этого оператора моделировать собственные псевдохеши. Вот более простой пример, реализующий объект как анонимный массив, но позволяющий ссылаться на него, как на хеш. Не пытайтесь обращаться с ним как с настоящим хешем; вы не сможете удалять из объекта пары ключ/значение посредством delete. Чтобы соединить обозначения массива и хеша, используйте настоящий псевдохеш.

```
package PsychoHash,

use overload "%{}" => \&as_hash;

sub as_hasn {
    my ($x) = shift;
    return { @$x };
}

sub new {
    my $class = shift,
    return bless [ @_ ] => $class;
}

$critter = new PsychoHash( height => 72, weight => 365, type => "camel" );

print $critter->{weight}; # выведет 365
```

См. также главу 14, где описывается механизм, позволяющий переопределять базовые операции над хешами, массивами и скалярами.

При перегрузке оператора постарайтесь не создавать объекты со ссылками на самих себя. Например:

```
use overload "+" => sub { bless [ \$_[0], \$_[1] ] };
```

Это провоцирует неприятности, поскольку если сказать $animal += $vegetable, то в результате $animal станет ссылкой на превращенный в объект массив, первым элементом которого является $animal. Это пример *циклической ссылки* (*circular reference*), которая ведет к тому, что даже при уничтожении $animal занимаемая ею память не будет освобождена, пока не завершит работу ваш процесс (или интерпретатор). См. раздел «Сборка мусора, циклические и слабые ссылки» в главе 8.

Интеллектуальное сопоставление

Ключ ~~ позволяет перегрузить логику интеллектуального сопоставления, используемую оператором ~~ и конструкцией given. См. раздел «Оператор интеллектуального сопоставления» в главе 3 и «Оператор given» в главе 4.

Перегруженная реализация оператора интеллектуального сопоставления не получает полный контроль над поведением механизма интеллектуального сопоставления, что не совсем обычно. В частности, в коде

```
package Foo;
use overload "~~" => "match";

my $obj = Foo->new();
$obj ~~ [ 1,2,3 ];
```

оператор интеллектуального сопоставления не произведет вызов:

```
$obj->match([1,2,3],0); # НЕВЕРНЫЙ ВЫЗОВ
```

Вместо этого Perl учтет приоритет операторов и будет поочередно сопоставлять $obj с каждым элементом массива в отдельности, пока не найдет соответствие. То есть, будет произведено три вызова метода:

```
$obj->match(1,0);
$obj->match(2,0);
$obj->match(3,0);
```

Подробности о том, когда вызывается перегруженная версия оператора интеллектуального сопоставления, приводятся в табл. 3.7 главы 3.

Конструктор копирования (=)

Несмотря на сходство с обычным оператором, = имеет особое и не вполне интуитивно понятное значение ключа перегрузки. Он *не* перегружает оператор присваивания. Он в принципе не может этого сделать, поскольку оператор присваивания должен быть зарезервирован для присваивания ссылкам, иначе все рухнет.

Обработчик для = используется в ситуациях, когда мутатор (mutator) (например, ++, -- или любой из операторов присваивания) применяется к ссылке, использующей свой объект совместно с другой ссылкой. Обработчик = позволяет программисту перехватить мутатор и самостоятельно скопировать объект, чтобы была изменена только копия. В противном случае будет изменен оригинал.

```
$copy = $original; # копирует только ссылку
++$copy;           # изменит объект, общий для двух ссылок
```

Теперь будьте к нам снисходительны. Предположим, что $original представляет собой ссылку на объект. Чтобы заставить ++$copy модифицировать только $copy, а не $original, сначала делается копия $copy, и переменной $copy присваивается ссылка на этот новый объект. Эта операция осуществляется, только когда выполняется ++$copy, поэтому $copy совпадает с $original до инкрементирования – но не после него. Иными словами, оператор ++ распознает необходимость копирования и обращается к конструктору копий.

Необходимость копирования распознают только мутаторы, такие как ++ или +=, либо nomethod, который описывается ниже. Если операция самогенерируется посредством +, как в:

```
$copy = $original;
$copy = $copy + 1;
```

то копирование не производится, потому что + не знает, что используется как мутатор.

Если во время выполнения некоторого мутатора потребуется вызвать конструктор копирования, но обработчик для = не задан, он может самогенерироваться как копирование строки, при условии что объект является обычным скаляром, а не чем-то более замысловатым.

Например, код, фактически выполняемый в этом случае:

```
$copy = $original;
...
++$copy;
```

может выглядеть примерно так:

```
$copy = $original;
...
$copy = $copy->clone(undef, "");
$copy->incr(undef, "");
```

При этом предполагается, что $original указывает на перегруженный объект, ++ был перегружен на \&incr, а = перегружен на \&clone.

Аналогичные действия запускаются при выполнении `$copy = $original++`, что интерпретируется как `$copy = $original; ++$original`.

Когда обработчик перегрузки отсутствует (nomethod и fallback)

Когда к объекту применяется не перегруженный оператор, Perl сначала пытается самогенерировать его поведение из других перегруженных операторов с помощью описанных выше правил. В случае неудачи Perl ищет реализацию перегрузки для nomethod, и, если она существует, использует ее. Этот обработчик является для операторов тем же, чем подпрограмма AUTOLOAD для подпрограмм: это то, что вы делаете, когда ничего другого придумать не можете.

Если использован ключ nomethod, за ним должна следовать ссылка на обработчик, принимающий четыре аргумента (а не три, как для всех остальных обработчиков). Первые три аргумента – такие же, как для любого другого обработчика; четвертым является строка, соответствующая оператору, обработчик которого отсутствует. Он служит той же цели, что и переменная $AUTOLOAD в подпрограммах AUTOLOAD.

Если Perl приходится искать обработчик для nomethod, а такого обработчика нет, возбуждается исключительная ситуация.

Чтобы избежать самогенерации или сделать так, чтобы при неудачной попытке самогенерации перегрузка вообще не производилась, можно определить специальный ключ перегрузки fallback (переход в аварийный режим). Используются три его состояния:

undef

: Если ключ fallback не установлен или явно установлен в undef, то последовательность событий при перегрузке не меняется: выполняется поиск обработчиков, производится попытка самогенерации и, наконец, вызывается обработчик nomethod. Если эта последовательность действий неудачна, возбуждается исключительная ситуация.

false

Если значение fallback установлено, но соответствует ложному значению (например, 0), самогенерация выполняться не будет. Perl вызовет обработчик nomethod, если он существует, и возбудит исключительную ситуацию в противном случае.

true

Обеспечивает почти такое же поведение, как undef, но при невозможности синтезировать обработчик с помощью самогенерации исключительная ситуация не возбуждается. Вместо этого Perl возвращается к применению неперегруженной версии оператора, как если бы прагмы use overload в этом классе вообще не было.

Перегрузка констант

Perl позволяет изменить способ интерпретации констант с помощью директивы overload::constant, которую удобнее поместить в метод import пакета. (При этом следует не забыть добавить директиву overload::remove_constant в метод unimport пакета, чтобы восстановить прежний порядок вещей.)

Обе директивы, overload::constant и overload::remove_constant, принимают списки пар ключ/значение. Ключами могут быть integer, float, binary, q и qr, а значениями – имена подпрограмм, анонимные подпрограммы или ссылки на код, которые будут обрабатывать константы.

```
sub import { overload::constant ( integer => \&integer_handler,
                                  float   => \&float_handler,
                                  binary  => \&base_handler,
                                  q       => \&string_handler,
                                  qr      => \&regex_handler ) }
```

Обработчики для integer и float будут вызываться лексическим анализатором Perl для числовых констант. Это происходит независимо от прагмы constant; простые инструкции, такие как:

```
$year = cube(12) + 1;        # целое
$pi   = 3.14159265358979;    # вещественное
```

будут вызывать указанные вами обработчики.

Ключ binary позволяет перехватывать двоичные, восьмеричные и шестнадцатеричные константы. q – обрабатывает строки в одинарных кавычках (в том числе строки, введенные с помощью q) и подстроки-константы в строках, заключаемых в qq и qx во внедренных документах (here documents). Наконец, qr обрабатывает неизменяющиеся участки в регулярных выражениях, как описано в конце главы 5.

Обработчик получает три аргумента. Первым является исходная константа в том виде, как она передается в Perl. Второй представляет результат интерпретации константы; например, 123_456 будет выглядеть как 123456.

Третий аргумент определяется только для строк, проходящих через обработчики q и qr, и может иметь одно из значений: qq, q, s или tr, в зависимости от того, как должна использоваться строка. Значение qq означает, что строка взята из интерполируемого контекста, например из двойных кавычек, обратных кавычек (backticks), оператора сопоставления m// или подстановки s///. Значение q означает,

что строка взята из неинтерполированного контекста. Значение s означает, что строка является строкой замены в операторе подстановки s///, а tr – что строка входит в состав выражения tr/// или y///.

Обработчик должен вернуть скаляр, который будет использоваться вместо константы. Часто этот скаляр является ссылкой на перегруженный объект, но ничто не мешает сделать что-либо более коварное:

```
package DigitDoubler;       # Модуль, помещаемый в DigitDoubler.pm
use overload;

sub import { overload::constant ( integer => \&handler,
                                  float   => \&handler ) }

sub handler {
    my ($orig, $interp, $context) = @_;
    return $interp * 2;     # удвоение значений всех констант
}

1;
```

Обратите внимание, что в двух ключах используется один и тот же обработчик, и это вполне оправданно. Теперь, сказав:

```
use DigitDoubler;

$trouble = 123;         # trouble получит значение 246
$jeopardy = 3.21;       # jeopardy получит значение 6.42
```

вы измените окружающий мир.

Если перехватываются строковые константы, рекомендуется также создать оператор конкатенации ("."), потому что интерполируемое выражение типа "ab$cd!!" является просто сокращенной записью более длинного 'ab' $cd !!'. Аналогично, отрицательные числа рассматриваются как отрицания положительных констант, поэтому нужно создать обработчик для neg, если перехватываются целые или действительные числа. (Раньше нам это не требовалось, потому что мы возвращали фактические числа, а не ссылки на перегруженные объекты.)

Обратите внимание, что overload::constant не распространяет свое действие на компиляцию внутри eval на этапе выполнения, что можно рассматривать как ошибку или функциональную особенность (bug или feature) в зависимости от точки зрения.

Открытые функции перегрузки

В Perl версии 5.6 прагма overload предоставляет следующие функции для общего пользования.

overload::StrVal(*OBJ*)

> Возвращает строковое значение, которое должен иметь *OBJ* в отсутствие перегрузки операции преобразования в строку ("").

overload::Overloaded(*OBJ*)

> Возвращает истинное значение, если для *OBJ* действует какая-либо перегрузка операторов, и ложное в противном случае.

overload::Method(*OBJ*, *OPERATOR*)

Возвращает ссылку на код, реализующий перегрузку оператора *OPERATOR*, когда тот выполняет действия над *OBJ*, или undef, если такой перегрузки не существует.

Наследование и перегрузка

Механизмы наследования и перегрузки имеют две точки соприкосновения. Во-первых, когда обработчик определяется как строка, а не как ссылка на код или анонимную подпрограмму, он интерпретируется как метод и может наследоваться от надклассов.

Во-вторых, любой класс, являющийся производным от перегруженного класса, в свою очередь подвергается этой перегрузке. Иными словами, перегрузка наследуется. Набор обработчиков в классе является рекурсивным объединением обработчиков во всех предках этого класса. Если обработчик присутствует в нескольких предках, фактически используемый обработчик определяется обычными правилами наследования. Например, если класс Alpha наследует классы Beta и Gamma в указанном порядке, и класс Beta перегружает + посредством \&Beta::plus_sub, а класс Gamma перегружает + строкой "plus_meth", при попытке применения + к объекту Alpha будет вызван метод Beta::plus_sub.

Поскольку значение ключа fallback не является обработчиком, его наследование не подчиняется указанным выше правилам. В текущей реализации Perl используется значение fallback для первого перегруженного предка, но этот выбор произволен и может быть изменен без уведомления (по крайней мере, без особого уведомления).

Перегрузка на этапе выполнения

Поскольку директивы use выполняются на этапе компиляции, единственный способ изменить перегрузку на этапе выполнения – использовать функцию eval:

```
eval " use overload '+' => \&my_add ";
```

Можно также сказать:

```
eval " no overload '+', '--', '<=' ";
```

хотя использование таких конструкций на этапе выполнения выглядит сомнительным.

Диагностика перегрузки

Если Perl скомпилирован с ключом -DDEBUGGING, при запуске программы с ключом *-Do* или его эквивалентом можно увидеть диагностические сообщения, касающиеся перегрузки. Узнать, какие операции перегружены, можно также с помощью команды m встроенного отладчика Perl.

Если вы сейчас ощущаете перегрузку, то следующая глава, возможно, расставит все по своим местам.

14

Связанные переменные

Некоторые человеческие начинания требуют маскировки. Иногда ее целью является обман, но чаще намерение заключается в передаче правдивой информации на более глубоком уровне. Например, приглашая соискателя на собеседование, интервьюер рассчитывает, что тот придет в галстуке, чтобы показать серьезную заинтересованность в получении работы, хотя оба знают, что он никогда не будет носить галстук на работе. Если вдуматься, то это странно: повязав на шею кусок ткани, можно чудесным образом получить работу. В культуре Perl оператор `tie` («связывать» или «галстук») играет схожую роль: он позволяет создать обычную с виду переменную, которая под своей личиной скрывает полноценный объект со своей собственной интересной личностью. Это просто некоторая магия, примерно как вытащить кролика из шляпы.

Скажем иначе, разыменовывающие символы `$`, `@`, `%` и `*` перед именем переменной многое говорят Perl и программирующим на нем: каждый из них предполагает определенный набор архетипов поведения. Эти архетипы можно изменять различными полезными способами с помощью `tie`, связывая переменные с классами, реализующими иные наборы режимов поведения. Например, можно создать обычный хеш и связать его (tie) с классом, превращающим хеш в базу данных, в результате чего при чтении из хеша Perl как по волшебству будет извлекать данные из внешнего файла базы данных, а при записи в хеш – сохранять данные во внешнем файле базы данных. В данном случае «как по волшебству» означает «незаметно сделать что-то очень сложное». Как гласит старое изречение, любая достаточно развитая технология неотличима от сценария Perl. (Без шуток, для тех, кто работает с внутренними механизмами Perl, *magic* (*волшебство*) – это технический термин, означающий любую дополнительную семантику применения таких переменных, как `%ENV` или `%SIG`. Связанные переменные – это всего лишь расширение такого волшебства.)

В Perl уже есть встроенные функции `dbmopen` и `dbmclose`, которые волшебным образом связывают переменные типа хеш с базами данных, однако они остались с тех времен, когда в Perl не было `tie`. Сейчас `tie` предоставляет более общий механизм. По правде сказать, Perl реализует `dbmopen` и `dbmclose` как раз посредством `tie`.

Скаляр, массив, хеш или дескриптор файла (через его typeglob) можно связать с любым классом, имеющим методы, которые перехватывают и эмулируют операции доступа к этим переменным. Первый из них вызывается в месте вызова самой функции tie: связывание переменной всегда вызывает конструктор для создания объекта, который Perl прячет от вас внутри «обычной» переменной. Впоследствии этот объект всегда можно извлечь с помощью функции tied:

```
tie VARIABLE, CLASSNAME, LIST;  # привязывает VARIABLE к CLASSNAME
$object = tied VARIABLE;
```

Эти две строки эквивалентны следующей:

```
$object = tie VARIABLE, CLASSNAME, LIST;
```

После связывания с обычной переменной можно работать как всегда, но при каждом обращении к ней автоматически будут вызываться методы спрятанного в ней объекта; вся сложная организация класса скрывается за этими вызовами методов. Если впоследствии потребуется разорвать связь между переменной и классом, можно отвязать переменную:

```
untie VARIABLE;
```

Функцию tie можно воспринимать как необычного вида bless, за исключением того, что «благословляется» голая переменная, а не ссылка на объект. Кроме того, функция tie может принимать дополнительные параметры, совсем как конструктор, что не очень удивляет, поскольку фактически она вызывает конструктор, имя которого зависит от типа связываемой переменной: TIESCALAR, TIEARRAY, TIEHASH или TIEHANDLE.[1] Эти конструкторы вызываются как методы класса с заданным *CLASSNAME* в качестве инвоканта и дополнительными аргументами, указанными в *LIST*. (Переменная *VARIABLE* не передается конструктору.)

Каждый из этих четырех конструкторов возвращает объект особым образом. Им, как и другим методам класса, безразлично, что они вызваны из tie, поскольку при желании их всегда можно вызвать непосредственно. В некотором смысле все волшебство заключено в tie, а не в классе, реализующем связывание. Что касается класса, то это обычный класс с необычными именами методов. (Кстати, некоторые связанные модули предоставляют дополнительные методы, невидимые через связанную переменную; эти методы должны вызываться явно, как любые другие методы объекта. Такие дополнительные методы могут предоставлять дополнительные услуги блокировки файлов, защиты транзакций и в принципе чего угодно, что может делать метод экземпляра объекта.)

Поэтому данные конструкторы вызывают bless и возвращают ссылку на объект, как любой другой конструктор. Эта ссылка не обязательно должна ссылаться на переменную того же типа, что и связываемая; она лишь должна пройти обработку функцией bless, чтобы связанная переменная могла найти обратную дорогу к вашему классу в поисках поддержки. Например, в нашем длинном примере с TIEARRAY будет использоваться объект на основе хеша, в котором удобно хранить дополнительную информацию об эмулируемом массиве.

---

[1] Поскольку конструкторы имеют непохожие имена, можно даже создать единый класс, который реализует их все. Это позволит связывать скаляры, массивы, хеши и дескрипторы файлов с одним и тем же классом, хотя обычно так не делают, поскольку это затрудняет написание других магических методов.

Функция tie не загрузит модуль вместо программиста с помощью use или require – он должен при необходимости сделать это самостоятельно, прежде чем вызвать tie. (С другой стороны, для обратной совместимости функция dbmopen попытается загрузить через use ту или иную реализацию модуля базы данных. Однако ее выбор можно отменить явным вызовом use, при условии, что загружаемый модуль присутствует в списке опробуемых функцией dbmopen. Более полное описание можно найти в электронной документации по модулю AnyDBM_File.)

Методы, вызываемые связанной переменной, имеют предопределенные имена, например FETCH и STORE, поскольку они вызываются неявно (т. е. в ответ на некоторые события) изнутри Perl. Эти имена записываются целиком ЗАГЛАВНЫМИ БУКВАМИ, что является распространенным соглашением для таких неявно вызываемых программ. (В число других специальных имен, следующих этому соглашению, входят BEGIN, CHECK, UNITCHECK, INIT, END, DESTROY и AUTOLOAD, а также UNIVERSAL->VERSION. На самом деле, почти все имена предопределенных переменных и дескрипторов файлов в Perl записываются буквами в верхнем регистре: STDIN, SUPER, CORE, CORE::GLOBAL, DATA, @EXPORT, @INC, @ISA, @ARGV и %ENV. Встроенные операторы и прагмы ударяются в другую крайность и вообще не используют заглавные буквы.)

Начнем с того, что расскажем о самом простом: как связать скалярную переменную.

Связывание скаляров

Чтобы иметь возможность связывать со скалярами, класс должен определять следующие методы: TIESCALAR, FETCH и STORE (возможно, также UNTIE и DESTROY). При связывании скалярной переменной вызывается метод TIESCALAR, при чтении скалярной переменной – метод FETCH, а в момент присваивания значения этой переменной вызывается STORE. Если сохранить объект, который вернет функция tie (или позже извлечь его с помощью tied), можно самостоятельно обращаться к лежащему в основе объекту: методы FETCH и STORE не будут при этом вызываться. В самом объекте никакого волшебства нет, это скорее предмет быта.

Метод UNTIE, если определен, вызывается при отвязывании переменной. Это дает возможность выполнить дополнительные действия или освободить ресурсы перед исчезновением связи и перед тем, как переменная превратится в обычную переменную.

Метод DESTROY, если определен, вызывается при исчезновении последней ссылки на связанный объект – как для всякого другого объекта. Это случается, когда программа завершает работу или вызывается функция untie, удаляющая ссылку, используемую tie. Однако untie не удаляет ожидающие обработки ссылки, которые могли быть помещены куда-то еще; вызов DESTROY тоже откладывается до того времени, когда исчезнут эти ссылки.

Пакеты Tie::Scalar и Tie::StdScalar из стандартного модуля Tie::Scalar предоставляют некоторые простые определения базового класса на случай, если вам не захочется определять эти методы самостоятельно. Tie::Scalar предоставляет простейшие методы, которые делают очень мало, а Tie::StdScalar предоставляет методы, благодаря которым связанный скаляр ведет себя как обычный скаляр Perl. (Это кажется совершенно бесполезным, но иногда требуется некоторая простая

обертка для обычной семантики скаляров – например, чтобы подсчитать, сколько раз выполнялось присваивание некоторой переменной.)

Прежде чем демонстрировать подробный пример и разбирать его механику, дадим вам маленький кусочек – для возбуждения аппетита и чтобы показать, как просто все на самом деле. Вот программа целиком:

```
#!/usr/bin/perl
package Centsible;
sub TIESCALAR { bless \my $self, shift }
sub STORE { ${ $_[0] } = $_[1] } # сделать дела по умолчанию
sub FETCH { sprintf "%.02f", ${ my $self = shift } } # округлить значение

package main;
tie $bucks, "Centsible";
$bucks = 45.00;
$bucks *= 1.0715; # налог
$bucks *= 1.0715; # обложить налогом повторно!
print "С вас, пожалуйста, $bucks.\n";
```

При выполнении этой программы выводится:

```
С вас, пожалуйста, 51.67.
```

Чтобы увидеть разницу, закомментируйте вызов `tie`, и вы получите:

```
С вас, пожалуйста, 51.66505125
```

Признаем, что работы здесь проделано больше, чем при обычном округлении чисел.

Методы связывания скаляров

Теперь, когда вы увидели, что нам предстоит, создадим более сложный класс для связывания скаляров. Мы не станем использовать в качестве базового класса готовый пакет (в основном за счет простоты скаляров), а вместо этого рассмотрим поочередно все четыре метода и создадим класс с именем `ScalarFile`. Скаляры, связанные с этим классом, содержат простые строки, и каждая такая переменная неявно связана с файлом, где хранится эта строка. (Переменные можно именовать так, чтобы их имена напоминали о том, на какие файлы мы ссылаемся.) Переменные связываются с классом так:

```
use ScalarFile;        # Загрузить ScalarFile.pm
tie $camel, "ScalarFile", "/tmp/camel.lot";
```

При связывании переменной ее прежнее содержимое уничтожается, а обычную семантику переменных замещает внутренняя связь между переменной и ее объектом. При запросе значения переменной `$camel` теперь будет читаться содержимое файла */tmp/camel.lot*, а в результате присваивания в файл */tmp/camel.lot* будет записываться новое значение, уничтожающее предшествующее.

Связывание выполняется для переменной, а не для значения переменной, поэтому природа связанной переменной не распространяется путем присваивания. Допустим, например, что копируется связанная переменная:

```
$dromedary = $camel;
```

Вместо того чтобы как обычно прочитать значение скалярной переменной $camel, Perl вызовет метод FETCH нижележащего ассоциированного объекта. Все происходит так, как если бы было написано:

```
$dromedary = (tied $camel)->FETCH():
```

Если сохранить объект, полученный при вызове tie, эту ссылку можно использовать непосредственно, как в следующем примере:

```
$clot = tie $camel, "ScalarFile", "/tmp/camel.lot";
$dromedary = $camel;          # через неявный интерфейс
$dromedary = $clot->FETCH(); # то же, но явно
```

Если класс реализует другие методы (помимо TIESCALAR, FETCH, STORE и DESTROY), их можно вызывать вручную при помощи $clot. Однако лучше заниматься своим делом и не соваться в нижележащий объект, поэтому можно часто видеть, что значение, полученное при вызове tie, игнорируется. Добраться до объекта, если он понадобился (например, если документация класса упоминает другие нужные вам методы), можно также вызовом tied. Игнорирование возвращаемого объекта также предотвращает некоторые виды ошибок, о которых рассказывается далее.

Вот преамбула нашего класса, которую мы поместим в *ScalarFile.pm*:

```
package ScalarFile:
use Carp;                # Деликатное распространение сообщений об ошибках.
use strict;              # Будем придерживаться некоторой дисциплины.
use warnings;            # Включение вывода предупреждений
                         # с лексической областью видимости
use warnings::register; # Разрешить пользователю сказать
                         # "use warnings 'ScalarFile'".
my $count = 0;           # Внутренний счетчик связанных объектов ScalarFile.
```

Стандартный модуль Carp экспортирует подпрограммы carp, croak и confess, которые будут использованы далее в примерах данного раздела. Как обычно, дополнительные подробности о Carp читайте в документации этого модуля.

В классе определены следующие методы.

CLASSNAME->TIESCALAR (*LIST*)

Метод класса TIESCALAR вызывается при связывании скалярной переменной. Необязательный аргумент *LIST* содержит параметры для правильной инициализации объекта. (В нашем примере есть только один параметр: имя файла.) Метод должен возвращать объект, но это не обязательно должна быть ссылка на скаляр. Однако в нашем примере это так:

```
sub TIESCALAR {           # в ScalarFile.pm
    my $class    = shift;
    my $filename = shift;
    $count++;             # Лексическая переменная с областью видимости
                          # в файле, закрытая в классе.
    return bless \$filename, $class;
}
```

Поскольку не существует скалярного эквивалента формирователей анонимных массивов и хешей, [] и {}, мы просто передаем функции bless объект ссылки лексической переменной, который становится анонимным, как только имя

выходит из области видимости. Это срабатывает (то же можно делать с массивами и хешами), если переменная действительно является лексической. Если попробовать проделать то же с глобальной переменной, может показаться, что все получилось, пока вы не попытаетесь создать еще один *camel.lot*. Не вздумайте написать что-нибудь подобное:

```
sub TIESCALAR { bless \$_[1], $_[0] } # НЕВЕРНО, если ссылка
                                      # на глобальную переменную.
```

Более надежный конструктор может проверять доступность файла. Сначала мы проверим доступность файла для чтения, поскольку не хотим уничтожить существующее значение. (Иными словами, не следует полагать, что пользователи сначала осуществляют запись. Они могут копить старые файлы Camel Lot от предыдущих прогонов программы.) Если нельзя открыть или создать файл с указанным именем, вежливо сообщим об ошибке, вернем undef и, возможно, выведем предупреждение средствами carp. (Можно использовать для этого и croak, это дело вкуса.) Попробуем определить, интересно ли пользователю наше предупреждение, при помощи прагмы warnings:

```
sub TIESCALAR {          # в ScalarFile.pm
    my $class    = shift;
    my $filename = shift;
    my $fh;
    if ( -r -w $filename ) {
        close $fh;
        $count++;
        return bless \$filename, $class;
    }
    carp "Невозможно связать $filename: $!" if warnings::enabled();
    return;
}
```

Теперь, имея конструктор, можно связать скаляр $string с файлом *camel.lot*:

```
tie ($string, "ScalarFile", "camel.lot") || die;
```

(Мы все еще делаем некоторые неразумные допущения. В производственной версии мы, вероятно, открыли бы дескриптор файла и запомнили его вместе с именем файла до конца действия связывания, заблокировав указатель для монопольного использования с помощью flock. В противном случае мы не защищены от ситуации гонок – см. раздел «Ошибки синхронизации» главы 20.)

SELF->FETCH

Этот метод вызывается при обращении к переменной (т. е. при чтении ее значения). Он не принимает аргументов помимо объекта, связанного с переменной. В нашем примере объект содержит имя файла.

```
sub FETCH {
    my $self = shift;
    confess "Я не метод класса" unless ref $self;
    return unless open my $fh, $$self;
    read($fh, my $value, -s $fh); # NB: не используйте -s с конвейерами!
    return $value;
}
```

На этот раз мы решили выйти из себя (возбудить исключительную ситуацию), если FETCH вернет что-либо, отличное от ссылки. (Когда он вызван как метод класса или по ошибке вызван как подпрограмма.) Другого способа вернуть ошибку нет, поэтому такой способ, вероятно, правильный. Perl все равно возбудил бы исключительную ситуацию, попытайся мы разыменовать $self; так что мы просто стараемся соблюдать вежливость и используем confess, чтобы извергнуть всю последовательность вызовов из стека на экран пользователя. (Если это можно считать вежливостью.)

Теперь мы можем увидеть содержимое *camel.lot*, сказав следующее:

```
tie($string, "ScalarFile", "camel.lot");
print $string;
```

SELF->STORE(*VALUE*)

Этот метод вызывается, когда переменной присваивается значение. Первый аргумент, *SELF*, всегда является объектом, связанным с переменной; *VALUE* является тем, что присваивается переменной. (Мы вольно обращаемся с термином «присваивается» – любая операция, модифицирующая переменную, может вызвать STORE.)

```
sub STORE {
    my($self,$value) = @_;
    ref $self                  || confess "не метод класса";
    open(my $fh, ">", $$self) || croak "нельзя разрушить $$self: $!";
    syswrite($fh, $value) == length $value
                               || croak "нельзя записать в $$self: $!";
    close($fh)                 || croak "нельзя закрыть $$self: $!";
    return $value;
}
```

После «присваивания» возвращается новое значение, поскольку это то, что делает присваивание. Если присваивание оказалось безуспешным, выводится сообщение посредством croak. Возможные причины для сбоев: отсутствие разрешения на запись в файл, отсутствие места на диске или нападение гремлинов на контроллер диска. Иногда вы управляете волшебством, а иногда волшебство управляет вами.

Теперь мы можем выполнить запись в *camel.lot*:

```
tie($string, "ScalarFile", "camel.lot");
$string  = "Вот первая строка camel.lot\n";
$string .= "А вот еще одна строка, дописанная автоматически.\n";
```

SELF->UNTIE

Этот метод вызывается функцией untie, и только untie. В данном примере он не используется, поэтому лишь покажем, как он вызывается:

```
sub UNTIE {
    my $self = shift;
    confess "Отвязано!";
}
```

См. предупреждение в разделе «Неочевидная ловушка при отвязывании».

SELF->DESTROY

Этот метод вызывается непосредственно перед тем, как объект, ассоциированный со связанной переменной, будет уничтожен сборщиком мусора, на случай, если необходимо сделать что-то особенное перед уничтожением. Как и для других классов, этот метод редко необходим, так как Perl автоматически освобождает память умирающего объекта. Наш метод DESTROY будет уменьшать счетчик связанных файлов:

```
sub DESTROY {
    my $self = shift;
    confess "Это не метод класса" unless ref $self;
    $count--;
}
```

Теперь можно создать дополнительный метод класса и для извлечения текущего значения счетчика. Фактически неважно, вызывается он как метод класса или как метод объекта, однако после вызова DESTROY у вас уже нет объекта, правда?

```
sub count {
    ### my $invocant = shift;
    $count;
}
```

В любой момент можно вызвать его как метод класса:

```
if (ScalarFile->count) {
    warn "Где-то еще есть связанные ScalarFiles. \n";
}
```

Это, пожалуй, все на данную тему. На самом деле, это даже больше чем все, поскольку мы реализовали ряд полезных особенностей, продиктованных соображениями полноты, надежности и общей эстетики (или отсутствия таковой). Конечно, можно создавать более простые классы TIESCALAR.

Переменные волшебных счетчиков

Вот простой класс Tie::Counter, созданный по образу и подобию одноименного модуля из CPAN. Переменные, связанные с этим классом, увеличиваются на 1 при каждом обращении к ним. Например:

```
tie my $counter, "Tie::Counter", 100;
@array = qw /Red Green Blue/;
for my $color (@array) {          # Выведет:
    print " $counter $color\n";   # 101 Red
}                                 # 102 Green
                                  # 103 Blue
```

Конструктор принимает в качестве необязательного аргумента начальное значение счетчика, по умолчанию равное 0. Присваивание счетчику устанавливает новое значение. Вот этот класс:

```
package Tie::Counter;
sub FETCH     { ++ ${ $_[0] } }
sub STORE     { ${ $_[0] } = $_[1] }
sub TIESCALAR {
```

```
        my ($class, $value) = @_;
        $value = 0 unless defined $value;
        bless \$value => $class;
    }
    1;    # если в модуле
```

Видите, какой он маленький? Чтобы построить такой класс, не требуется много кода.

Обход значений в цикле

Благодаря волшебству связывания, массив может вести себя как скаляр. Интерфейс tie способен преобразовывать скалярный интерфейс в интерфейс массива. Модуль Tie::Cycle из архива CPAN использует скаляр для обхода в цикле значений в массиве. Объект запоминает текущее положение и перемещается на один шаг вперед при каждом обращении. По достижении конца он автоматически возвращается в начало:

```
package Tie::Cycle;

sub TIESCALAR {
    my $class    = shift;
    my $list_ref = shift;
    return unless ref $list_ref eq ref [];
    my @shallow_copy = map { $_ } @$list_ref;
    my $self = [ 0, scalar @shallow_copy, \@shallow_copy ];
    bless $self, $class;
}

sub FETCH {
    my $self  = shift;
    my $index = $$self[0]++;
    $$self[0] %= $self->[1];
    return $self->[2]->[ $index ];
}

sub STORE {
    my $self     = shift;
    my $list_ref = shift;
    return unless ref $list_ref eq ref [];
    $self = [ 0, scalar @$list_ref, $list_ref ]
}
```

Этот объект удобно использовать для поочередного применения классов CSS в смежных строках в HTML-таблице без усложнения кода:

```
tie my $row_class, "Tie::Cycle", [ qw(odd even) ];

for my $item (@items) {
    print qq(<tr class="$row_class">...</tr>);
}
```

Точно так же легко можно было бы увеличить число классов CSS, не изменяя основной код программы:

```
tie my $row_class, "Tie::Cycle", [ qw(red green blue) ],
```

Волшебное изгнание $_

Следующий любопытно-экзотический класс, underscore[1], позволяет сделать незаконными нелокализованные применения $_. Вместо директивы use, вызывающей метод класса import, этот модуль должен загружаться директивой no, чтобы вызвать редко используемый метод unimport (см. главу 11). Пользователь говорит:

```
no underscore;
```

и после этого любое использование $_ как нелокализованной глобальной переменной будет возбуждать исключительную ситуацию.

Вот маленький набор тестов для модуля:

```
#!/usr/bin/perl
no underscore;
@tests = (
    "Присваивание" => sub { $_ = "Bad" }
    "Чтение"     => sub { print },
    "Сопоставление"   => sub { $x = /badness/ },
    "Отсечение"       => sub { chop },
    "Проверка файла"   => sub { -x },
    "Вложенность"     => sub { for (1..3) { print } },
);

while ( ($name, $code) = splice(@tests, 0, 2) ) {
    print "Проверяю $name: ";
    eval { &$code };
    print $@ ? "обнаружено" : " отсутствует!";
    print "\n";
}
```

который выведет следующее:

```
Проверяю Присваивание: обнаружено
Проверяю Чтение: обнаружено
Проверяю Сопоставление: обнаружено
Проверяю Отсечение: обнаружено
Проверяю Проверка файла: обнаружено
Проверяю Вложенность: 123 отсутствует!
```

В последнем случае «отсутствует» получено благодаря должной локализации в цикле for и потому безопасному доступу.

А вот и сам любопытно-экзотический модуль underscore. (Мы ведь говорили, что он любопытно-экзотический?) Он работает, поскольку волшебство связывания надежно скрыто с помощью local. Модуль вызывает tie в методе инициализации, поэтому require тоже работает.

```
package underscore;
use warnings;
```

[1] Любопытно отметить, что класс underscore появился в качестве примера в одном из первых изданий этой книги, затем перекочевал в книгу «Perl Cookbook», вдохновившись которой, Дэн Когай (Dan Kogai) создал модуль CPAN.

```
use strict;
use Carp ();
our $VERSION = sprintf "%d.%02d", q$Revision: 0.1 $ =~ /(\d+)/g;

sub TIESCALAR {
    my ($pkg, $code, $msg) = @_;
    bless [$code, $msg], $pkg;
}

sub unimport {
    my $pkg    = shift;
    my $action = shift;
    no strict "refs";
    my $code = ref $action
                 ? $action
                 : ($action
                     ? \&{ "Carp::" . $action }
                     : \&Carp::croak
                   );
    my $msg = shift || "$_ is forbidden";
    untie $_ if tied $_;
    tie $_, __PACKAGE__, $code, $msg;
}

sub import{ untie $_ }

sub FETCH{ $_[0]->[0]($_[0]->[1]) }
sub STORE{ $_[0]->[0]($_[0]->[1]) }

1;    # Конец underscore
```

Тяжело с пользой смешивать вызовы use и no для этого класса в программе, поскольку все они происходят на этапе компиляции, а не на этапе выполнения. Можно было бы непосредственно вызвать Underscore->import и Underscore->unimport, как это делают use и no. Но обычно, чтобы отречься от своих слов и позволить себе снова свободно использовать $_, следует применить к этой переменной local, в чем и заключалась задумка.

Связывание массивов

В классе, реализующем связанный массив, должны быть определены по крайней мере методы TIEARRAY, FETCH и STORE. Есть много необязательных методов: среди них, разумеется, вездесущие UNTIE и DESTROY, а также методы STORESIZE и FETCHSIZE, обеспечивающие доступ к $#array и scalar(@array). Кроме того, когда Perl необходимо очистить массив, вызывается CLEAR, а когда Perl требуется упреждающе выделить дополнительную память для настоящего массива, вызывается EXTEND.

Можно также определить методы POP, PUSH, SHIFT, UNSHIFT, SPLICE, DELETE и EXISTS, если требуется, чтобы соответствующие функции Perl работали со связанным массивом. Класс Tie::Array может служить базовым для реализации первых пяти из этих функций в терминах FETCH и STORE. (Стандартная реализация методов DELETE и EXISTS в Tie::Array просто вызывает croak.) Если FETCH и STORE определены, тип структуры данных, содержащейся в объекте, не имеет значения.

С другой стороны, класс Tie::StdArray (определен в стандартном модуле Tie::Array) – это базовый класс с методами по умолчанию, которые предполагают, что объект содержит обычный массив. Вот простой класс связываемого массива, использующий Tie::StdArray, благодаря чему ему требуется определить только методы, реализующие нестандартное поведение.

```
#!/usr/bin/perl
package ClockArray;
use Tie::Array;
our @ISA = "Tie::StdArray";
sub FETCH {
    my($self,$place) = @_;
    $self->[ $place % 12 ];
}

sub STORE {
    my($self,$place,$value) = @_;
    $self->[ $place % 12 ] = $value;
}

package main;
tie my @array, "ClockArray";
@array = ( "a" ... "z" );
print "@array\n";
```

Если запустить эту программу, она выведет "y z o p q r s t u v w x". Этот класс представляет массив с дюжиной элементов, подобно циферблату часов, пронумерованных от 0 до 11. Если запросить пятнадцатый элемент массива, вы получите третий. Считайте это вспомогательным средством для тех, кто не научился читать время в 24-часовом формате.

Методы связывания массивов

Это был простой способ. Теперь некоторые важные бытовые подробности. Для их демонстрации создадим массив, границы которого фиксируются при его создании. Если попытаться обратиться к элементу за пределами этих границ, возбуждается исключительная ситуация. Например:

```
use BoundedArray;
tie @array, "BoundedArray", 2;

$array[0] = "отлично";
$array[1] = "хорошо";
$array[2] = "великолепно";
$array[3] = "тпру!"; # Запрещено: выводится сообщение об ошибке.
```

Код преамбулы для этого класса таков:

```
package BoundedArray;
use Carp;
use strict;
```

Чтобы в дальнейшем избежать необходимости определять метод SPLICE, унаследуем класс Tie::Array:

```
use Tie::Array;
our @ISA = ("Tie::Array");
```

CLASSNAME->TIEARRAY(*LIST*)

Будучи конструктором класса, метод TIEARRAY должен возвращать «освященную» ссылку, через которую будет эмулироваться связанный массив.

В следующем примере, чтобы показать, что *в действительности* не обязательно возвращать ссылку на массив, для представления нашего объекта была выбрана ссылка на хеш. Хеш хорошо подходит на роль обобщенного типа данных: элемент хеша с ключом "BOUND" будет хранить индекс верхней границы, а элемент с ключом "DATA" будет хранить фактические данные. Если кто-то попытается разыменовать возвращаемый объект вне пределов класса (несомненно, считая, что это ссылка на массив), возникнет исключительная ситуация.

```
sub TIEARRAY {
    my $class = shift;
    my $bound = shift;
    confess "usage: tie(\@ary, 'BoundedArray', max_subscript)"
        if @_ || $bound =~ /\D/;
    return bless { BOUND => $bound, DATA => [] }, $class;
}
```

Теперь можно сказать:

```
tie(@array, "BoundedArray", 3); # допускается максимальный индекс 3
```

и гарантировать, что в массиве будет не более четырех элементов. При чтении или записи отдельного элемента массива вызываются FETCH и STORE, как для скаляров, но с дополнительным аргументом, содержащим индекс.

SELF->FETCH(*INDEX*)

Этот метод вызывается при обращении к отдельному элементу связанного массива. Помимо ссылки на объект он получает еще один аргумент: индекс элемента, значение которого мы пытаемся извлечь.

```
sub FETCH {
    my ($self, $index) = @_;
    if ($index > $self->{BOUND}) {
        confess "Array OOB: $index > $self->{BOUND}";
    }
    return $self->{DATA}[$index];
}
```

SELF->STORE(*INDEX, VALUE*)

Этот метод вызывается, когда присваивается значение элементу связанного массива. Помимо ссылки на объект он принимает два аргумента: индекс элемента и значение, которое мы пытаемся присвоить. Например:

```
sub STORE {
    my($self, $index, $value) = @_;
    if ($index > $self->{BOUND} ) {
        confess "Array OOB: $index > $self->{BOUND}";
    }
    return $self->{DATA}[$index] = $value;
}
```

SELF->UNTIE

Этот метод вызывается подпрограммой untie. В данном примере он не потребовался. См. предупреждение в разделе «Неочевидная ловушка при отвязывании» ниже.

SELF->DESTROY

Perl вызывает этот метод, когда нужно уничтожить связанную переменную и освободить занимаемую ею память. Это почти никогда не требуется в языке со сборкой мусора, поэтому в данном примере мы этот метод опустим.

SELF->FETCHSIZE

Метод FETCHSIZE должен вернуть общее число элементов связанного массива, ассоциированного с *SELF*. Это эквивалент вызова scalar(@array), который обычно дает $#array + 1.

```
sub FETCHSIZE {
    my $self = shift;
    return scalar @{$self->{DATA}};
}
```

SELF->STORESIZE(*COUNT*)

Этот метод устанавливает общее число элементов связанного массива, ассоциированного с *SELF*, равным *COUNT*. Если массив уменьшается, нужно удалить элементы за пределами *COUNT*. Если массив увеличивается, нужно добавить новые элементы с неопределенными значениями. Класс BoundedArray также гарантирует, что массив не выйдет за изначально установленные границы.

```
sub STORESIZE {
    my ($self, $count) = @_;
    if ($count > $self->{BOUND}) {
        confess "Array OOB: $count > $self->{BOUND}";
    }
    $#{$self->{DATA}} = $count;
}
```

SELF->EXTEND(*COUNT*)

С помощью метода EXTEND Perl сообщает, что размер массива, вероятно, будет увеличен до COUNT элементов. Благодаря этому можно выделить один непрерывный участок памяти вместо множества маленьких. Поскольку объекты BoundedArray имеют фиксированную верхнюю границу размера массива, мы не станем определять этот метод.

SELF->EXISTS(*INDEX*)

Этот метод проверяет наличие элемента в позиции *INDEX* связанного массива. В нашем классе BoundedArray мы просто используем встроенную функцию exists, предварительно убедившись, что это не попытка заглянуть дальше фиксированной верхней границы.

```
sub EXISTS {
    my ($self, $index) = @_;
    if ($index > $self->{BOUND}) {
        confess "Array OOB: $index > $self->{BOUND}";
    }
```

```
        exists $self->{DATA}[$index],
    }
```

SELF->DELETE(*INDEX*)

Метод DELETE **удаляет элемент в позиции** *INDEX* **из связанного массива** *SELF*. **В классе** BoundedArray **этот метод выглядит почти идентично** EXISTS, **но это отклонение от нормы.**

```
    sub DELETE {
        my ($self  $index) = @_;
        print STDERR "deleting!\n",
        if ($index > $self->{BOUND}) {
            confess "Array OOB: $index > $self->{BOUND}";
        }
        delete $self->{DATA}[$index],
    }
```

SELF->CLEAR

Метод вызывается, чтобы очистить массив, например, когда массиву присваивается список новых значений (или пустой список), но не тогда, когда он передается функции undef. **Поскольку очищенный** BoundedArray **всегда удовлетворяет условию верхней границы, проверка здесь не нужна:**

```
    sub CLEAR {
        my $self = shift;
        $self->{DATA} = [];
    }
```

Если присвоить массиву список, метод CLEAR **будет вызван. но он не получит список значений. Поэтому при таком нарушении верхней границы:**

```
    tie(@array, "BoundedArray", 2);
    @array = (1, 2, 3, 4),
```

метод CLEAR **выполнится успешно. Исключительная ситуация возникнет при последующем вызове** STORE. **Операция присваивания один раз вызовет метод** CLEAR **и четыре раза – метод** STORE.

SELF->PUSH(*LIST*)

Этот метод добавляет в конец массива элементы из *LIST*. **Вот как он может быть реализован в нашем классе** BoundedArray:

```
    sub PUSH {
        my $self = shift;
        if (@_ + $#{$self->{DATA}} > $self->{BOUND}) {
            confess "Попытка протолкнуть слишком много элементов"
        }
        push @{$self->{DATA}}, @_;
    }
```

SELF->UNSHIFT(*LIST*)

Этот метод вставляет элементы *LIST* **в начало массива. Для нашего класса** BoundedArray **эта подпрограмма аналогична** PUSH.

SELF->POP

Метод POP удаляет из массива последний элемент и возвращает его. Для BoundedArray это метод-однострочник:

```
sub POP { my $self = shift; pop @{$self->{DATA}}; }
```

SELF->SHIFT

Метод SHIFT удаляет из массива первый элемент и возвращает его. Для BoundedArray он аналогичен POP.

SELF->SPLICE(*OFFSET*, *LENGTH*, *LIST*)

Этот метод позволяет сращивать массив *SELF*. Для имитации splice – встроенной функции Perl – аргумент *OFFSET* является необязательным, по умолчанию равным нулю, причем смещения с отрицательными значениями откладываются от конца массива. Аргумент *LENGTH* тоже должен быть необязательным и по умолчанию содержать длину оставшейся части массива. *LIST* может быть пустым. При правильной имитации встроенной функции метод должен возвращать список из *LENGTH* элементов исходного массива, начиная с *OFFSET* (т.е. список элементов, заменяемых на *LIST*).

Поскольку сращивание представляет собой достаточно сложную операцию, мы вообще не будем ее определять, а просто воспользуемся подпрограммой SPLICE из модуля Tie::Array, которая нам досталась в результате наследования Tie::Array. При этом SPLICE будет определена на основе других методов BoundedArray, поэтому проверка границ сохранится.

Этим завершается наш класс BoundedArray. Он лишь немного искажает семантику массивов. Но мы можем сделать и лучше, и гораздо более экономно.

Удобство обозначений

Одним из замечательных свойств переменных является возможность их интерполяции. Одним из не столь замечательных свойств функций является отсутствие возможности интерполяции. Связанный массив позволяет создать интерполируемую функцию. Допустим, необходимо вставить в строку случайные числа. Можно просто сказать:

```
#!/usr/bin/perl
package RandInterp;
sub TIEARRAY { bless \my $self }
sub FETCH { int rand $_[1] };

package main;
tie @rand, "RandInterp";
for (1,10,100,1000) {
    print "Случайным числом, меньшим $_, является $rand[$_]\n";
}
$rand[32] = 5; # Это отформатирует наш системный диск?
```

Если запустить эту программу, она выведет:

```
Случайным числом, меньшим 1, является 0
Случайным числом, меньшим 10, является 3
Случайным числом, меньшим 100, является 46
Случайным числом, меньшим 1000, является 755
```

```
Can't locate object method "STORE" via package "RandInterp" at foo line 10.
[Невозможно найти метод "STORE" объекта через пакет "RandInterp"]
```

Как видите, нет большой беды, что мы даже не реализовали STORE. Просто, как обычно, возникает исключительная ситуация.

Связывание хешей

Класс, реализующий связанный хеш, должен определить восемь методов. TIEHASH создает новые объекты. FETCH и STORE обеспечивают доступ к парам ключ/значение. EXISTS проверяет присутствие ключа в хеше, а DELETE удаляет ключ вместе с его значением.[1] CLEAR очищает хеш, удаляя все пары ключ/значение. FIRSTKEY и NEXTKEY осуществляют обход пар ключ/значение при вызове keys, values или each. И как всегда, если нужно выполнить какие-то особые действия при удалении объекта, можно определить метод DESTROY. (Если вам кажется, что методов слишком много, значит, вы невнимательно прочли последний раздел, посвященный массивам. Как бы там ни было, можете смело наследовать методы по умолчанию из стандартного модуля Tie::Hash, переопределяя только те, которые вас интересуют. Кроме того, Tie::StdHash предполагает, что реализация тоже является хешем.)

Допустим, например, что нужно создать хеш, в котором при каждом присваивании значения ключу будет происходить не перезапись прежнего значения, а добавление нового значения в конец массива значений. То есть, когда вы говорите:

```
$h{$k} = "один";
$h{$k} = "два";
```

на самом деле выполняется:

```
push @{ $h{$k} }, "один";
push @{ $h{$k} }, "два";
```

Это не очень сложная идея, поэтому модуль должен получиться очень простым. Если в качестве базового класса выбрать класс Tie::StdHash, то так и будет. Вот класс Tie::AppendHash, который это реализует:

```
package Tie::AppendHash;
use Tie::Hash;
our @ISA = ("Tie::StdHash");
sub STORE {
    my ($self, $key, $value) = @_;
    push @{$self->{key}}, $value;
}
1;
```

Методы связывания хешей

Вот пример интересного класса связанного хеша: он предоставляет хеш, в котором содержатся dot-файлы[2] конкретного пользователя (т.е. файлы, имена которых

---

[1] Напомним, что Perl различает ключи, отсутствующие в хеше, и ключи со значением undef. Проверить эти две ситуации можно с помощью exists и defined соответственно.

[2] Иногда их называют скрытыми, потому что по умолчанию команда ls не выводит их. – *Прим. перев.*

начинаются с точки, как принято именовать файлы настроек в UNIX). Вы обращаетесь к хешу, передавая в качестве индекса имя файла (без точки), и получаете содержимое соответствующего dot-файла. Например:

```
use DotFiles;
tie %dot, "DotFiles";
if ( $dot{profile} =~ /MANPATH/ ||
     $dot{login}   =~ /MANPATH/ ||
     $dot{cshrc}   =~ /MANPATH/    ) {
    print "Кажется. вам нужно установить переменную MANPATH\n";
}
```

Вот еще один способ использования нашего связанного класса:

```
# Третьим аргументом является имя пользователя,
# к dot-файлам которого мы привяжемся.
tie %him, "DotFiles", "daemon".
foreach $f (keys %him) {
    printf "dot-файл демона %s имеет размер %d\n", $f, length $him{$f};
}
```

В примере DotFiles **мы реализуем объект как обычный хеш, содержащий несколько важных полей, из которых только поле** {CONTENTS} **содержит то, что пользователь считает хешем. Фактические поля объекта перечислены в табл. 14.1.**

Таблица 14.1. Поля объекта в примере DotFiles

Поле	**Содержание**
USER	**Владелец dot-файлов, представляемых объектом.**
HOME	**Расположение этих dot-файлов.**
CLOBBER	**Разрешено ли изменять и удалять эти dot-файлы.**
CONTENTS	**Хеш имен dot-файлов и отображений содержимого.**

Вот начало *DotFiles.pm*:

```
package DotFiles;
use Carp;
sub whowasi { (caller(1))[3]
my $DEBUG = 0;
sub debug { $DEBUG = @_ ? shift   1 }
```

В данном примере мы хотим иметь возможность включить вывод отладочной информации, которая облегчит трассировку во время разработки, для чего устанавливаем переменную $DEBUG. **Мы также поддерживаем удобную внутреннюю функцию для вывода предупреждений:** whowasi **возвращает имя функции, вызвавшей текущую («дедушки»** whowasi**).**

Вот методы связанного хеша DotFiles**:**

CLASSNAME->TIEHASH(*LIST*)

Конструктор DotFiles**:**

```
sub TIEHASH {
    my $self   = snift;
```

```
    my $user   = shift || $>,
    my $dotdir = shift || "";

    croak "usage: @{[ &whowasi ]} [USER [DOTDIR]]" if @_;

    $user = getpwuid($user) if $user =~ /^\d+$/;
    my $dir = (getpwnam($user))[7]
            || croak "@{ [&whowasi] }: no user $user"
    $dir .= "/$dotdir" if $dotdir;

    my $node = {
        USER     => $user,
        HOME     => $dir,
        CONTENTS => {},
        CLOBBER  => 0,
    };

    opendir(DIR, $dir)
            || croak "@{[&whowasi]}: can't opendir $dir: $!";
    for my $dot ( grep /^\./ && -f "$dir/$_", readdir(DIR)) {
        $dot =~ s/^\.//;
        $node->{CONTENTS}{$dot} = undef;
    }
    closedir DIR;

    return bless $node, $self;
}
```

Стоит, вероятно, отметить, что если мы собираемся применять проверки файлов к значениям, возвращаемым readdir, следует добавлять перед ними соответствующий каталог (как мы и делаем). Иначе, без вызова chdir, может оказаться, что проверяется не тот файл.

SELF->FETCH(*KEY*)

Этот метод реализует чтение элемента из связанного хеша. Помимо ссылки на объект он принимает один аргумент: ключ, значение которого мы хотим получить. Ключ является строкой, и с ним можно делать все, что можно делать со строкой.

Вот пример извлечения значения элемента для нашего DotFiles:

```
sub FETCH {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $dir = $self->{HOME};
    my $file = "$dir/.$dot";

    unless (exists $self->{CONTENTS}->{$dot} || -f $file) {
        carp "@{[&whowasi]}: no $dot file" if $DEBUG;
        return undef;
    }

    # Реализовать кэш.
    if (defined $self->{CONTENTS}->{$dot}) {
```

```
        return $self->{CONTENTS}->{$dot},
    } else {
        return $self->{CONTENTS}->{$dot} = `cat $dir/.$dot`;
    }
}
```

Мы немного схитрили, выполнив команду UNIX *cat*(1), но более переносимой (и более эффективной) была бы реализация, самостоятельно открывающая файл и выполняющая его чтение. С другой стороны, поскольку dot-файлы являются особенностью UNIX, мы не очень этим озабочены. Или не должны быть озабочены. Или что-то в таком духе...

SELF->STORE(*KEY*, *VALUE*)

Этот метод выполняет всю черную работу при установке (записи) элемента в связанном хеше. Помимо ссылки на объект он принимает два аргумента: ключ и новое значение.

В нашем примере DotFiles **мы разрешим пользователям перезаписать файл лишь после вызова метода** clobber **с исходным объектом, полученным от** tie:

```
sub STORE {
    carp &whowasi if $DEBUG;
    my $self  = shift;
    my $dot   = shift;
    my $value = shift;
    my $file  = $self->{HOME} . "/.$dot";

    croak "@{[&wnowasi]}: $file невозможно перезаписать"
        unless $self->{CLOBBER};
    open(F, "> $file") || croak "не могу открыть $file: $!"
    print F $value;
    close(F);          || croak "не могу закрыть $file: $!"
}
```

Если кому-то потребуется изменить файл, можно сказать:

```
$ob = tie %daemon_dots, 'daemon"
$ob->clobber(1);
$daemon_dots{signature} = "Настоящий демон\n";
```

Другим вариантом является установка {CLOBBER} **посредством** tied:

```
tie %daemon_dots, "DotFiles", "daemon"
tied(%daemon_dots)->clobber(1);
```

или в одну команду:

```
(tie %daemon_dots, "DotFiles", "daemon")->clobber(1);
```

Метод clobber **определяется просто:**

```
sub clobber {
    my $self = shift;
    $self->{CLOBBER} = @_ ? shift : 1;
}
```

SELF->DELETE(*KEY*)

Этот метод обрабатывает запросы на удаление элемента из хеша. Если ваш эмулированный хеш использует настоящий хеш, можете вызвать настоящую функцию delete. Кроме того, мы постараемся проверить, действительно ли пользователь хочет уничтожить файл:

```
sub DELETE {
    carp &whowasi if $DEBUG;
    my $self = shift,
    my $dot  = shift;
    my $file = $self->{HOME} . "/.$dot";
    croak "@{[&whowasi]}. не стану удалять файл $file"
        unless $self->{CLOBBER};
    delete $self->{CONTENTS}->{$dot};
    unlink $file || carp "@{[&whowasi]} не могу удалить $file: $!":
}
```

SELF->CLEAR

Этот метод вызывается, когда нужно полностью очистить хеш, обычно в случае присваивания ему пустого списка. В нашем примере при этом удаляются все dot-файлы пользователя! Это настолько опасно, что мы потребуем, чтобы значение CLOBBER было больше 1, прежде чем выполнить это действие:

```
sub CLEAR {
    carp &whowasi if $DEBUG;
    my $self = shift;
    croak "@{[&whowasi]}: не буду удалять все dot-файлы $self->{USER}"
        unless $self->{CLOBBER} > 1;
    for my $dot ( keys %{$self->{CONTENTS}}) {
        $self->DELETE($dot);
    }
}
```

SELF->EXISTS(*KEY*)

Этот метод запускается при вызове пользователем функции exists с данным хешем. В нашем примере мы будем искать ответ в элементе хеша {CONTENTS}:

```
sub EXISTS {
    carp &whowasi if $DEBUG;
    my $self = shift·
    my $dot  = shift;
    return exists $self->{CONTENTS}->{$dot}
}
```

SELF->FIRSTKEY

Этот метод вызывается, когда пользователь начинает обход хеша, например, при вызове keys, values или each. В результате вызова keys в скалярном контексте мы сбрасываем внутреннее состояние, чтобы очередной вызов each в инструкции return возвращал первый ключ.

```
sub FIRSTKEY {
    carp &whowasi if $DEBUG;
    my $self = shift;
```

```
        my $temp = keys %{$self->{CONTENTS}};
        return scalar each %{$self->{CONTENTS}};
    }
```

SELF->NEXTKEY(*PREVKEY*)

Этот метод служит итератором для функций keys, values или each. *PREVKEY* является последним запрошенным ключом и предоставляется Perl. Это удобно, если методу *NEXTKEY* нужно знать свое предыдущее состояние для вычисления последующего.

В нашем примере для представления данных связанного хеша используется настоящий хеш, только этот хеш хранится в поле *CONTENTS* хеша, а не в самом хеше. Поэтому мы можем просто положиться на итератор each:

```
    sub NEXTKEY {
        carp &whowasi if $DEBUG;
        my $self = shift;
        return scalar each %{ $self->{CONTENTS} }
    }
```

SELF->UNTIE

Этот метод вызывается подпрограммой untie. В этом примере он не нужен. См. предупреждение в разделе «Неочевидная ловушка при отвязывании».

SELF->DESTROY

Этот метод вызывается, когда объект связанного хеша должен быть удален из памяти. Практической нужды в нем нет, кроме как для отладки и дополнительной подчистки. Вот очень простой вариант:

```
    sub DESTROY {
        carp &whowasi if $DEBUG;
    }
```

Теперь, когда мы показали все эти методы, вот вам домашнее задание: вернуться назад, найти все точки интерполяции @{[&whowasi]} и заменить их простым связанным скаляром с именем $whowasi, осуществляющим то же самое.

Связывание дескрипторов файлов

Класс, реализующий связанный дескриптор файла, должен определить следующие методы: TIEHANDLE и по крайней мере один из PRINT, PRINTF, WRITE, READLINE, GETC и READ. Класс может также предоставить метод DESTROY и методы BINMODE, OPEN, CLOSE, EOF, FILENO, SEEK, TELL, READ и WRITE, чтобы разрешить применение соответствующих встроенных функций. (А если говорить более точно, то WRITE соответствует syswrite и не имеет никакого отношения к встроенной функции write для вывода со спецификациями форматирования.)

Связанные дескрипторы файлов особенно удобны, когда Perl встраивается в другую программу (например, Apache или *vi*) и вывод в STDOUT или STDERR требуется перенаправлять особым образом.

Однако дескрипторы файлов совсем не обязательно связывать с файлами. Инструкции вывода можно использовать для создания структур данных в памяти,

а команды ввода – для чтения их в программе. Вот простой способ обратить порядок следования вызовов `print` **и** `printf`**, не меняя при этом местами отдельные строчки:**

```
package ReversePrint 0.01 {
    use strict;
    sub TIEHANDLE {
        my $class = shift;
        bless [], $class;
    }

    sub PRINT {
        my $self = shift;
        push @$self, join("" => @_);
    }

    sub PRINTF {
        my $self = shift;
        my $fmt = shift;
        push @$self, sprintf($fmt, @_);
    }

    sub READLINE {
        my $self = shift;
        pop @$self;
    }
}

my $m = "--MORE--\n";
tie *REV, "ReversePrint"

# Выполнить несколько print и printf.
print REV "The fox is now dead.$m";

printf REV <<"END", int rand 10000000;
The quick brown fox jumps
over the lazy dog %d times!
END

print REV <<"END";
The quick brown fox jumps
over the lazy dog.
END

# Теперь обратное чтение из того же дескриптора
print while <REV>;
```

В результате будет выведено:

```
The quick brown fox jumps
over the lazy dog.
The quick brown fox jumps
over the lazy dog 3179357 times!
The fox is now dead.--MORE--
```

Методы привязки дескрипторов файлов

В качестве примера посложнее создадим дескриптор файла, преобразующий выводимые строки в верхний регистр. Исключительно для собственного удовольствия мы будем начинать файл тегом <SHOUT> при его открытии и заканчивать его тегом </SHOUT> при закрытии. Благодаря этому мы можем выражаться на корректном (well-formed) языке XML.

Вот начало файла *Shout.pm* с реализацией класса:

```
package Shout;
use Carp;                # Чтобы выводить ошибки посредством croak
```

Теперь перечислим определения методов в *Shout.pm*.

CLASSNAME->TIEHANDLE(*LIST*)

Это конструктор класса, возвращающий ссылку, как обычно, обработанную функцией bless.

```
sub TIEHANDLE {
    my $class = shift;
    my $form = shift;
    open(my $self, $form, @_)  || croak "не могу открыть $form@_: $!";
    if ($form =~ />/) {
        print $self "<SHOUT>\n";
        $$self->{WRITING} = 1;    # Не забыть вывести закрывающий тег
    }
    return bless $self, $class;   # $self является ссылкой на glob
}
```

Здесь мы открываем новый дескриптор файла в соответствии с режимом и именем файла, переданными оператору tie, выводим в файл <SHOUT> и возвращаем ссылку на него, обработанную функцией bless. В вызове open происходит много интересного, но мы отметим лишь, что в дополнение к обычной идиоме «open or die» в аргументе my $self функции open передается неопределенный скаляр, который open умеет автовивифицировать в typeglob. Тот факт, что это typeglob, тоже важен, потому что typeglob содержит не только действительный объект ввода/вывода файла, но и другие удобные (и бесплатные, с точки зрения реализации) структуры данных, например, скаляр ($$$self), массив (@$$self) и хеш (%$$self). (Не говоря уже о подпрограмме, &$$self.)

$form является аргументом, содержащим имя файла или режим. Если это имя файла, то массив @_ пуст, и выполняется вызов open с двумя аргументами. В противном случае $form указывает режим для остальных аргументов.

После вызова open проверяется, нужно ли выводить открывающий тег. Если да, то мы это делаем. И сразу после этого используем одну из упомянутых выше глобальных структур данных. Выражение $$self->{WRITING} служит примером использования глобальных переменных для хранения интересной информации. В данном случае мы запоминаем, выводился ли открывающий тег, чтобы знать, нужно ли выводить соответствующий закрывающий тег. Мы используем хеш %$$self, чтобы дать полю приличествующее имя. Можно было бы использовать скаляр $$$self, но это не было бы самодокументацией. (Или было бы *только* само (self) документацией, в зависимости от того, как на это посмотреть.)

SELF->PRINT(*LIST*)

Реализует вывод в связанный дескриптор. *LIST* содержит все, что передается функции print. **В нашем методе все элементы списка *LIST* переводятся в верхний регистр:**

```
sub PRINT {
    my $self = shift;
    print $self map {uc} @_;
}
```

SELF->READLINE

Этот метод возвращает данные, когда чтение из дескриптора осуществляется через оператор угловых скобок (<FH>**) или** readline. **Метод должен возвращать** undef, **если данных больше нет.**

```
sub READLINE {
    my $self = shift;
    return <$self>;
}
```

Здесь мы просто возвращаем <$self>, **чтобы метод вел себя должным образом при вызове в скалярном или списочном контексте.**

SELF->GETC

Этот метод вызывается, когда к связанному дескриптору файла применяется getc.

```
sub GETC {
    my $self = shift;
    return getc($self);
}
```

Как и некоторые другие методы нашего класса Shout, **метод** GETC **просто вызывает соответствующую встроенную функцию и возвращает результат**

SELF->OPEN(*LIST*)

Наш метод TIEHANDLE **открывает файл самостоятельно, но программа, использующая класс** Shout **и вызывающая при этом** open, **реально вызывает вот этот метод.**

```
sub OPEN {
    my $self = shift;
    my $form = shift;
    my $name = "$form@_";
    $self->CLOSE;
    open($self, $form, @_)      || croak "невозможно вновь открыть $name: $!";
    if ($form =~ />/) {
        print $self "<SHOUT>\n" || croak "невозможно начать вывод: $!";
        $$self->{WRITING} = 1;  # Не забыть вывести закрывающий тег
    }
    else {
        $$self->{WRITING} = 0;  # Не выводить закрывающий тег
    }
    return 1;
}
```

Мы вызываем собственный метод CLOSE, чтобы явно закрыть файл, если пользователь об этом не позаботился. Затем открываем новый файл с именем, указанным в open, и выводим в него открывающий тег, если необходимо.

SELF->CLOSE

Этот метод обрабатывает запрос на закрытие файла. Мы ищем конец файла и в случае успеха выводим </SHOUT>, а затем вызываем встроенную функцию close.

```
sub CLOSE {
    my $self = shift;
    if ($$self->{WRITING}) {
        $self->SEEK(0, 2)           || return;
        $self->PRINT("</SHOUT>\n") || return;
    }
    return close $self;
}
```

SELF->SEEK(*LIST*)

При передаче связанного дескриптора файла функции seek вызывается метод SEEK.

```
sub SEEK {
    my $self = shift;
    my ($offset, $whence) = @_;
    return seek($self, $offset, $whence);
}
```

SELF->TELL

Вызывается при передаче связанного дескриптора файла функции tell.

```
sub TELL {
    my $self = shift;
    return tell $self;
}
```

SELF->PRINTF(*LIST*)

Вызывается при передаче связанного дескриптора файла функции printf. Список *LIST* содержит формат и элементы, подлежащие выводу.

```
sub PRINTF {
    my $self = shift;
    my $template = shift;
    return $self->PRINT(sprintf $template, @_);
}
```

Здесь мы применяем sprintf для генерации форматированной строки и передачи ее PRINT для перевода в верхний регистр. Однако использовать встроенную функцию sprint не обязательно. Можно предусмотреть собственную интерпретацию спецификаторов формата, начинающихся символом процента.

SELF->READ(*LIST*)

Отвечает за выполнение операции чтения при вызове read или sysread. Обратите внимание, что мы модифицируем первый аргумент *LIST* «по месту», имитируя способность read заполнять скаляр, передаваемый во втором аргументе.

```
sub READ {
    my ($self, undef, $length, $offset) = @_;
    my $bufref = \$_[1];
    return read($self, $$bufref, $length, $offset);
}
```

SELF->WRITE(*LIST*)

Вызывается при записи в дескриптор с помощью syswrite. Здесь мы преобразуем символы записываемой строки в верхний регистр.

```
sub WRITE {
    my $self = shift;
    my $string = uc(shift);
    my $length = shift || length $string;
    my $offset = shift || 0;
    return syswrite $self, $string, $length, $offset;
}
```

SELF->EOF

Возвращает булево значение, когда дескриптор файла, связанный с классом Shout, проверяется на состояние конца файла посредством eof.

```
sub EOF {
    my $self = shift;
    return eof $self;
}
```

SELF->BINMODE(*IOLAYER*)

Определяет уровень ввода/вывода для использования с дескриптором файла. Когда уровень не задан, дескриптор файла переводится в двоичный режим (уровень :raw), если файловая система различает текстовые и двоичные файлы.

```
sub BINMODE {
    my $self = shift;
    my $disc = shift || ":raw";
    return binmode $self, $disc;
}
```

Так следовало бы написать, но в нашем случае это бесполезно, поскольку функция open уже произвела запись в дескриптор. Поэтому в данном случае следовало бы, вероятно, сказать:

```
sub BINMODE { croak("Слишком поздно использовать binmode") }
```

SELF->FILENO

Этот метод должен возвращать дескриптор файла (fileno), ассоциированный операционной системой со связанным дескриптором файла.

```
sub FILENO {
    my $self = shift;
    return fileno $self;
}
```

SELF->UNTIE

Вызывается подпрограммой untie. В данном примере он не потребовался. См. предупреждение в разделе «Неочевидная ловушка при отвязывании».

SELF->DESTROY

Как и в других типах связывания, этот метод вызывается, когда связанный объект должен быть уничтожен. Он позволяет объекту прибрать за собой. Здесь мы закрываем файл, если программа забыла его закрыть. Мы могли бы просто сказать close $self, **но лучше вызвать метод класса** CLOSE. **Тогда, если разработчик класса решит изменить способ закрытия файлов, этот метод** DESTROY **не придется модифицировать.**

```
sub DESTROY {
    my $self = shift;
    $self->CLOSE; # Закрыть файл с помощью метода CLOSE класса Shout
}
```

Вот демонстрация класса Shout**:**

```
#!/usr/bin/perl
use Shout;
tie(*FOO, Shout::, ">filename");
print FOO "hello\n";           # Выведет HELLO.
seek FOO, 0, 0;                # Переход в начало файла
@lines = <FOO>;                # Вызовет метод READLINE
close FOO;                     # Закрыть файл явно.
open(FOO, "+<", "filename"); # Повторно открыть FOO, вызвав OPEN
seek(FOO, 8, 0);               # Перешагнуть через "<SHOUT>\n"
sysread(FOO, $inbuf, 5);       # Прочитать 5 байтов из FOO в $inbuf
print "found $inbuf\n";        # Должен вывести"hello".
seek(FOO, -5, 1);              # Вернуться назад к началу "hello".
syswrite(FOO, "ciao!\n", 6); # Записать 6 байтов в FOO
untie(*FOO);                   # Неявно вызовет метод CLOSE.
```

После выполнения этого фрагмента файл будет содержать:

```
<SHOUT>
CIAO!
</SHOUT>
```

Вот еще некоторые странные и удивительные вещи, относящиеся к этой внутренней глобальной переменной. Мы используем тот же хеш, что и раньше, но с новыми ключами PATHNAME **и** DEBUG. **Сначала перегрузим операцию преобразования в строку, чтобы при выводе наших объектов отображался путь (см. главу 13):**

```
# Ну, это круто!
use overload q("") => sub { $_[0]->pathname }

# Это заглушка для добавления во все трассируемые функции
sub trace {
    my $self = shift;
    local $Carp::CarpLevel = 1;
    Carp::cluck("\ntrace magical method") if $self->debug;
}

# Перегрузка обработчика для вывода пути.
sub pathname {
    my $self = shift;
    confess "я не метод класса" unless ref $self;
```

```
        $$self->{PATHNAME} = shift if @_;
        return $$self->{PATHNAME};
    }
    # Два режима.
    sub debug {
        my $self = shift;
        my $var = ref $self ? \$$self->{DEBUG}   \our $Debug;
        $$var = shift if @_;
        return ref $self ? $$self->{DEBUG} || $Debug : $Debug;
    }
```

А затем будем вызывать `trace` **при входе во все обычные методы, например:**

```
    sub GETC { $_[0]->trace; # НОВИНКА
        my($self) = @_;
        getc($self);
    }
```

А также сохраним строку пути в `TIEHANDLE` **и** `OPEN`**:**

```
    sub TIEHANDLE {
        my $class = shift;
        my $form  = shift;
        my $name  = "$form@_";      # НОВИНКА
        open(my $self, $form, @_) || croak "невозможно открыть $name  $!";
        if ($form =~ />/) {
            print $self "<SHOUT>\n";
            $$self->{WRITING} = 1; # Не забыть вывести конечный тег
        }
        bless $self, $class;        # $fh является ссылкой на glob
        $self->pathname($name);     # НОВИНКА
        return $self;
    }

    sub OPEN { $_[0]->trace;        # НОВИНКА
        my $self = shift;
        my $form = shift;
        my $name = "$form@_
        $self->CLOSE;
        open($self, $form, @_)   croak  невозможно вновь открыть. $
        $self->pathname($name);     # НОВИНКА
        if ($form =~ />/) {
            print($self "<SHOUT>\n") || croak "невозможно начать вывод: $!
            $$self->{WRITING} = 1; # Не забыть вывести конечный тег
        }
        else {
            $$self->{WRITING} = 0; # Запомнить, что не нужно выводить конечный тег
        }
        return 1;
    }
```

Где-то нужно также вызвать `$self->debug(1)` **для включения отладки. После этого все вызовы** `Carp::cluck` **будут выводить осмысленные сообщения. Ниже показано, что мы получим при повторном открытии файла выше. Здесь видно, что мы опустились на три уровня вложенности при закрытии старого файла и подготовке к открытию нового:**

```
trace magical method at foo line 87
    Shout::SEEK('>filename', '>filename', 0, 2) called at foo line 81
    Shout::CLOSE('>filename') called at foo line 65
    Shout::OPEN('>filename', '+<', 'filename') called at foo line 141
```

Созидающие дескрипторы файлов

Один и тот же дескриптор файла может быть связан как со вводом, так и с выводом двунаправленного канала. Допустим, вы хотите запустить программу *bc*(1) (калькулятор произвольной точности) таким способом:

```
use Tie::Open2;

tie *CALC, "Tie::Open2", "bc -l",
$sum = 2;
for (1 .. 7) {
    print CALC "$sum * $sum\n";
    $sum = <CALC>;
    print "$_  $sum";
    chomp $sum;
}
close CALC;
```

Можно предположить, что будет выведено следующее:

```
1: 4
2: 16
3: 256
4: 65536
5: 4294967296
6: 18446744073709551616
7: 340282366920938463463374607431768211456
```

Ожидания оправданные, будь на компьютере программа *bc*(1) и определи мы Tie::Open2 так, как описано ниже. На этот раз в качестве внутреннего объекта будет использоваться массив. Он содержит два фактических файловых дескриптора для чтения и записи. (Черную работу открытия двунаправленного канала выполняет метод IPC::Open2; мы занимаемся только приятными вещами.)

```
package Tie::Open2;
use strict;
use Carp;
use Tie::Handle;              # не наследуйте этот класс!
use IPC::Open2;

sub TIEHANDLE {
    my ($class, @cmd) = @_;
    no warnings "once",
    my @fhpair = \do { local(*RDR, *WTR) };
    bless $_, "Tie::StdHandle" for @fhpair;
    bless(\@fhpair => $class)->OPEN(@cmd) || die;
    return \@fhpair;
}
```

```
sub OPEN {
    my ($self, @cmd) = @_;
    $self->CLOSE if grep {defined} @{ $self->FILENO }
    open2(@$self, @cmd);
}

sub FILENO {
    my $self = shift;
    [ map { fileno $self->[$_] } 0,1 ];
}

for my $outmeth ( qw(PRINT PRINTF WRITE) ) {
    no strict "refs";
    *$outmeth = sub {
        my $self = shift;
        $self->[1]->$outmeth(@_);
    };
}
for my $inmeth ( qw(READ READLINE GETC) ) {
    no strict "refs";
    *$inmeth = sub {
        my $self = shift;
        $self->[0]->$inmeth(@_);
    };
}
for my $doppelmeth ( qw(BINMODE CLOSE EOF)) {
    no strict "refs";
    *$doppelmeth = sub {
        my $self = shift;
        $self->[0]->$doppelmeth(@_) && $self->[1]->$doppelmeth(@_);
    };
}
for my $deadmeth ( qw(SEEK TELL)) {
    no strict "refs";
    *$deadmeth = sub {
        croak("can't $deadmeth a pipe");
    };
}
1;
```

Последние четыре цикла, по нашему мнению, просто чрезвычайно элегантны. За разъяснением происходящего вернитесь к разделу «Замыкания в качестве шаблонов функций» главы 8.

Вот еще более безумный набор классов. Имена пакетов подскажут, для чего эти классы служат.

```
use strict;
package Tie::DevNull;

    sub TIEHANDLE {
        my $class = shift;
        my $fh = local *FH;
        bless \$fh, $class;
    }
```

```
    for (qw(READ READLINE GETC PRINT PRINTF WRITE)) {
        no strict "refs";
        *$_ = sub { return };
    }

package Tie::DevRandom;

    sub READLINE { rand() . "\n"; }
    sub TIEHANDLE {
        my $class = shift;
        my $fh = local *FH;
        bless \$fh, $class;
    }

    sub FETCH { rand() }
    sub TIESCALAR {
        my $class = shift;
        bless \my $self, $class;
    }

package Tie::Tee;

    sub TIEHANDLE {
        my $class = shift;
        my @handles;
        for my $path (@_) {
            open(my $fh, ">$path") || die "невозможно вывести в $path";
            push @handles, $fh;
        }
        bless \@handles, $class;
    }

    sub PRINT {
        my $self = shift;
        my $ok = 0;
        for my $fh (@$self) {
            $ok += print $fh @_;
        }
        return $ok == @$self
    }
```

Класс Tie::Tee **эмулирует стандартную программу UNIX *tee*(1), которая отправляет один выходной поток по нескольким различным адресам. Класс** Tie::DevNull **эмулирует устройство */dev/null* в UNIX-системах. А класс** Tie::DevRandom **создает случайные числа в виде дескриптора файла или скаляра, в зависимости от того, вызываете вы** TIEHANDLE **или** TIESCALAR! **Вот как осуществляется обращение к ним:**

```
package main;

tie *SCATTER,  "Tie::Tee", qw(tmp1 - tmp2 >tmp3 tmp4);
tie *RANDOM,   "Tie::DevRandom";
tie *NULL,     "Tie::DevNull";
tie my $randy, "Tie::DevRandom";
```

```
for my $i (1..10) {
    my $line = <RANDOM>;
    chomp $line;
    for my $fh (*NULL, *SCATTER) {
        print $fh "$i: $line $randy\n";
    }
}
```

При этом на экран будет выведено что-нибудь вроде следующего:

```
1: 0.124115571686165 0.20872819474074
2: 0.156618299751194 0.678171662366353
3: 0.799749050426126 0.300184963960792
4: 0.599474551447884 0.213935286029916
5: 0.700232143543861 0.800773751296671
6: 0.201203608274334 0.0654303290639575
7: 0.605381294683365 0.718162304090487
8: 0.452976481105495 0.574026269121667
9: 0.736819876983848 0.391737610662044
10: 0.518606540417331 0.381805078272308
```

Но это не все! Вывод на экран был произведен благодаря символу «-» в строке `tie *SCATTER` **выше. Но в той же строке было выдвинуто требование создать файлы *tmp1*, *tmp2* и *tmp4*, а файл *tmp3* открыть в режиме добавления в конец. (В цикле также производился вывод в дескриптор файла** `*NULL`, **но это не произвело никакого интересного эффекта, если, конечно, для вас не представляют интереса черные дыры.)**

Неочевидная ловушка при отвязывании

Когда используется объект, полученный от `tie` **или** `tied`, **а класс определяет метод-деструктор, возникает неочевидная ловушка, которой вы должны беречься. Рассмотрим такой (возможно, натянутый) пример класса, использующего файл для регистрации всех значений, присваиваемых скаляру:**

```
package Remember;

sub TIESCALAR {
    my $class = shift;
    my $filename = shift;
    open(my $handle, ">", $filename)
        || die "Невозможно открыть $filename: $!\n";
    print $handle "Старт\n";
    bless {FH => $handle, VALUE => 0}, $class;
}

sub FETCH {
    my $self = shift;
    return $self->{VALUE};
}

sub STORE {
    my $self = shift;
    my $value = shift;
    my $handle = $self->{FH};
```

```
        print $nandle "$value\n";
        $self->{VALUE} = $value;
    }

    sub DESTROY {
        my $self = shift;
        my $handle = $self->{FH};
        print $handle "Конец\n";
        close $handle;
    }

    1;
```

А это пример использования класса Remember**:**

```
use strict;
use Remember;

my $fred;
$x = tie $fred, "Remember", "camel.log";
$fred = 1;
$fred = 4;
$fred = 5;
untie $fred;
system "cat camel.log";
```

А это вывод примера:

```
Старт
1
4
5
Конец
```

Пока все хорошо. Добавим в класс Remember **еще один метод, разрешающий комментарии в файле, например такой:**

```
sub comment {
    my $self = shift;
    my $message = shift;
    print { $self->{FH} } $handle $message, "\n";
}
```

И вот предыдущий пример с добавлением вызова метода comment**:**

```
use strict;
use Remember;

my ($fred, $x);
$x = tie $fred, "Remember", "camel.log";
$fred = 1;
$fred = 4;
comment $x "changing...";
$fred = 5;
untie $fred;
system "cat camel.log";
```

Теперь файл окажется пустым, чего вы, вероятно, не желали. И вот почему это происходит. Связывание переменной ассоциирует ее с объектом, полученным от конструктора. На этот объект обычно имеется только одна ссылка: та, что скрывается за самой связанной переменной. Вызов `untie` разрывает ассоциацию и ликвидирует эту ссылку. Поскольку ссылок на объект больше не осталось, запускается метод `DESTROY`. Однако в примере выше мы имели вторую ссылку на объект, связанную с `$x`. Это значит, что после `untie` по-прежнему сохраняется действующая ссылка на объект. `DESTROY` запущен не будет, запись буфера в файл и закрытие файла не произойдет. Вот почему не было вывода: буфер дескриптора файла по-прежнему находился в памяти. На диск он не попадет до выхода из программы.

Чтобы убедиться в этом, можно использовать флаг командной строки *-w* или включить в текущую лексическую область видимости прагму `use warnings "untie"`. В том и другом случае выявляется обращение к `untie` на момент существования ссылок на связанный объект. В этом случае Perl выводит предупреждение:

```
untie attempted while 1 inner references still exist
[попытка untie при существовании 1 внутренней ссылки]
```

Чтобы заставить программу работать правильно и подавить вывод предупреждающих сообщений, ликвидируйте лишние ссылки на связанный объект *перед* вызовом `untie`. Это можно сделать явно:

```
undef $x;
untie $fred;
```

Однако часто эта проблема решается простым выходом из области видимости переменных в нужный момент.

Модули для связывания в CPAN

Прежде чем вы загоритесь написать собственный модуль для связывания, убедитесь, что никто не сделал этого до вас. В CPAN имеется масса модулей для связывания, и число их растет с каждым днем. (Или, во всяком случае, с каждым месяцем.) В табл. 14.2 перечислены некоторые из них.

Таблица 14.2. Модули для связывания в CPAN

Модуль	Описание
`IO::WrapTie`	Оборачивает связанные объекты в интерфейс `IO::Handle`
`MLDBM`	Прозрачное хранение сложных данных, а не только простых строк, в файле DBM
`Tie::Cache::LRU`	Реализует кэш с алгоритмом вытеснения давно неиспользуемых
`Tie::Const`	Предоставляет скаляры-константы и хеши-константы
`Tie::Counter`	Заколдованные скалярные переменные, увеличивающиеся при каждом обращении
`Tie::CPHash`	Реализует хеш, не чувствительный к регистру, но сохраняющий его
`Tie::Cycle`	Циклический обход значений в списке через скаляр
`Tie::DBI`	Связывает хеши с реляционными базами данных DBI

Таблица 14.2 (продолжение)

Модуль	Описание
Tie::Dict	Связывает хеш с сервером RPC dict
Tie::DictFile	Связывает хеш с локальным файлом словаря
Tie::DNS	Связывает интерфейс с Net::DNS
Tie::EncryptedHash	Хеши (и производные объекты) с шифрованием полей
Tie::FileLRUCache	Реализует облегченный, базирующийся на файловой системе, постоянный кэш LRU (с вытеснением давно неиспользуемых)
Tie::FlipFlop	Реализует связывание, чередующее два значения
Tie::HashDefaults	Позволяет хешу иметь значения по умолчанию
Tie::HashHistory	Отслеживает историю всех изменений в хеше
Tie::iCal	Связывает хеши с файлами iCal
Tie::IxHash	Упорядоченные ассоциативные массивы для Perl
Tie::LDAP	Реализует интерфейс к базе данных LDAP
Tie::Persistent	Предоставляет долгоживущие структуры данных через tie
Tie::Pick	Случайным образом выбирает (и удаляет) элементы множеств
Tie::RDBM	Связывает хеши с реляционными базами данных
Tie::STDERR	Направляет вывод дескриптора STDERR в другой процесс, например в почтовую программу
Tie::Syslog	Связывает дескриптор файла для автоматического вывода в системный журнал
Tie::TextDir	Связывает каталог файлов
Tie::Toggle	Поочередно возвращает истинное и ложное значения, и так до бесконечности
Tie::TZ	Связывает $TZ, настройки %ENV и вызов *tzset*(3)
Tie::VecArray	Интерфейс массива для вектора битов
Tie::Watch	Устанавливает контрольные точки на переменные Perl
Win32::TieRegistry	Предоставляет мощные и простые способы работы с реестром Microsoft Windows

III

Perl как технология

15

Межпроцессные взаимодействия

У процессов в компьютере почти столько же способов общения друг с другом, сколько у людей. Сложность взаимодействий между процессами не следует недооценивать. Бесполезно искать смысл в словах, если ваш приятель говорит на языке жестов. Аналогично, два процесса могут общаться между собой, только договорившись о средствах связи и основываясь на соглашениях, построенных на этих средствах. Как и при любом обмене информацией, соглашения, о которых нужно договориться, простираются от лексических до практических: от используемого жаргона до очередности вступления в разговор. Эти соглашения необходимы, потому что очень трудно передавать голую семантику в отсутствие контекста.

На нашем жаргоне межпроцессные взаимодействия обычно называются IPC (Interprocess Communication). Perl реализует различные механизмы IPC-взаимодействия, от самых простых до самых сложных. Выбор механизма зависит от сложности передаваемой информации. В простейших случаях почти никакая информация не передается: просто поступает уведомление о том, что в некоторый момент произошло некоторое событие. В Perl передача информации о простых событиях производится с помощью механизма *сигналов*, сделанного по образцу системы сигналов UNIX.

Средства поддержки сокетов в Perl – противоположная крайность. Эти средства позволяют обмениваться информацией с любым другим процессом в Интернете в соответствии с выбранным протоколом, который реализуют обе стороны. Конечно, за такую свободу приходится платить: нужно пройти через ряд этапов, чтобы установить соединение и обеспечить использование языка, понятного другому процессу. Это, в свою очередь, может потребовать соблюдения некоторого количества других странных обычаев, зависящих от местных нравов. Для выполнения всех требований протокола вас могут даже обязать разговаривать на одном из таких языков, как XML, Java или Perl. Ужас.

Посередине располагаются средства, предназначенные, в основном, для связи с процессами на той же машине. В их число входят старые добрые файлы, конвейеры, очереди FIFO и различные системные вызовы System V IPC. Поддержка этих механизмов зависит от платформы; современные UNIX-системы (в том числе

Apple OS X) должны поддерживать все средства, а большинство средств, за исключением сигналов и SysV IPC, поддерживается во всех свежих операционных системах Microsoft, включая каналы, ветвление процессов (forking), блокировку файлов и сокеты.[1]

Дополнительные сведения о переносимости в целом можно найти в стандартном наборе документации Perl (в любом формате, поддерживаемом вашей системой), в разделе *perlport*. Специфическую для Microsoft информацию можно найти в разделах *perlwin32* и *perlfork*, которые устанавливаются отнюдь не только на системах, созданных Microsoft. В качестве учебников предлагаем следующие книги:

- «Perl Cookbook, Second Edition», Tom Christiansen и Nathan Torkington, O'Reilly, главы с 16 по 18.
- «Advanced Programming in the UNIX Environment», by W. Richard Stevens and Stephen A. Rago, Addison-Wesley (3rd Edition, May 2013)[2].
- «TCP/IP Illustrated», W. Richard Stevens, тома I–III (Addison-Wesley, 1992–1996).

Сигналы

В Perl применяется простая модель обработки сигналов: хеш `%SIG` содержит ссылки (символические или жесткие) на определенные пользователем обработчики сигналов. Некоторые события побуждают операционную систему доставлять сигналы затронутому ими процессу. Обработчик, соответствующий этому событию, вызывается с одним аргументом, содержащим название сигнала. Для отправки сигнала другому процессу используется функция `kill`. Можно считать это передачей процессу одного бита информации.[3] Если в том процессе для данного сигнала установлен обработчик, то при получении сигнала он может выполнить некоторый код. Однако передающий процесс не может получить какой-либо ответ, кроме известия о том, что сигнал был успешно отправлен. Отправитель не имеет никакой обратной связи, чтобы узнать, что сделал с сигналом принимающий процесс и сделал ли вообще.

Мы классифицировали эту возможность как форму IPC, но на практике сигналы могут поступать из разных источников, не только от других процессов. Сигнал может поступить и от вашего собственного процесса или быть сгенерирован в результате того, что пользователь ввел особую клавиатурную последовательность (например, Control-C или Control-Z), или исходить от ядра системы вследствие возникновения особого события, такого, например, как завершение работы порожденного процесса, исчерпание памяти стека вашим процессом, превышение ограничения размера файла или памяти. Однако вашему собственному процессу нелегко различить эти случаи. Сигнал – это как посылка, таинственным образом оказавшаяся на вашем пороге и не имеющая обратного адреса. Вскрывать ее следует осторожно.

---

1 За исключением сокетов `AF_UNIX`.

2 У. Ричард Стивенс и Стивен Раго «UNIX. Профессиональное программирование», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2014.

3 На практике это скорее пять или шесть битов, в зависимости от того, сколько сигналов определяет операционная система, и от того, использует ли процесс-собеседник факт отсутствия дополнительного сигнала.

Поскольку элементами массива %SIG могут быть жесткие ссылки, в качестве обработчиков простых сигналов общепринято использовать анонимные функции:

```
$SIG{INT}  = sub { die "\nВыметаюсь!\n" };
$SIG{ALRM} = sub { die "Сработал твой будильник" };
```

Можно также создать именованную функцию и присвоить ее имя или ссылку на нее соответствующему элементу хеша. Например, чтобы перехватить сигналы прерывания и завершения (часто привязываемые к Control-C и Control-\ на клавиатуре), установите такой обработчик:

```
sub catch_zap {
    my $signame = shift;
    our $shucks++;
    die "Кто-то послал мне SIG$signame!";
}
$shucks = 0;
$SIG{INT}  = "catch_zap"; # всегда означает &main::catch_zap
$SIG{INT}  = \&catch_zap; # самый лучший вариант
$SIG{QUIT} = \&catch_zap; # перехват еще одного сигнала
```

Обратите внимание, что в обработчике сигналов мы лишь устанавливаем глобальную переменную и инициализируем исключительную ситуацию вызовом die. Это было важно в старых версиях Perl, когда еще не были реализованы безопасные сигналы, потому что в большинстве систем библиотека C нереентерабельна, а сигналы передаются асинхронно. Это могло вызвать ошибку даже в самом надежном коде на Perl. С реализацией безопасных сигналов эта проблема исчезла.

Еще более простой способ перехвата сигналов состоит в использовании прагмы sigtrap для установки простых обработчиков сигналов по умолчанию:

```
use sigtrap qw(die INT QUIT);
use sigtrap qw(die untrapped normal-signals
    stack-trace any error-signals);
```

Эта прагма полезна, если вы не хотите утруждаться созданием собственного обработчика, но все же хотите перехватывать опасные сигналы и осуществлять контроль над завершением работы. По умолчанию некоторые из этих сигналов могут быть настолько смертельными для процесса, что программа просто замрет на месте, получив один из них. К несчастью, это означает, что все функции END для обработки при завершении и методы DESTROY для закрытия объектов не будут вызваны. Зато они вызываются при обычных исключительных ситуациях Perl (например, при вызове die), поэтому данную прагму можно использовать для безболезненного преобразования сигналов в исключительные ситуации. Даже если сами вы не будете обрабатывать сигналы, ваша программа поведет себя корректно. Читайте в описании use sigtrap в главе 29 о многих дополнительных возможностях этой прагмы.

Обработчику в %SIG можно также присвоить одну из строк, "IGNORE" или "DEFAULT", и тогда Perl попытается отбросить сигнал или разрешить для него действие по умолчанию (хотя некоторые сигналы нельзя ни перехватить, ни игнорировать, например сигналы KILL и STOP; список сигналов, имеющихся в системе, и соответствующие сигналам режимы по умолчанию можно найти на странице *signal*(3) справочного руководства, если оно у вас есть).

В операционной системе сигналы представлены как числа, но Perl, как и большинство людей, предпочитает символические имена волшебным числам. Чтобы узнать имена сигналов, можно вывести ключи хеша %SIG или выполнить команду *kill -l*, если она поддерживается системой. Можно также использовать стандартный модуль Perl Config, чтобы определить соответствие между именами и номерами сигналов в операционной системе. Пример см. в *Config*(3).

Поскольку %SIG – это глобальный хеш, изменения в нем влияют на всю программу. Часто более тактично по отношению к остальной части программы заключить свою обработку сигналов в ограниченную область видимости. Сделайте это при помощи обработчика сигнала local, который прекратит свое действие после выхода из охватывающего блока. (Но помните, что значения local видны в функциях, вызываемых внутри этого блока.)

```
{
    local $SIG{INT} = 'IGNORE';
    ...       # Делайте здесь, что хотите, игнорируя все SIGINT
    fn();     # SIGINT игнорируются внутри fn() тоже!
    ...       # И здесь.
}             # Выход из блока восстанавливает прежнее значение $SIG{INT}.

fn();         # SIGINT не игнорируются внутри fn() (предположительно).
```

Передача сигналов группе процессов

Процессы (по крайней мере, в UNIX) объединяются в группы, обычно соответствующие заданию в целом. Например, при запуске одной команды оболочки, состоящей из последовательности команд-фильтров, передающих данные по конвейеру одна другой, все эти процессы (и порожденные ими процессы) будут принадлежать одной и той же группе процессов. Этой группе процессов будет назначен номер, соответствующий номеру процесса лидера группы. Если, посылая сигнал, указать положительный номер процесса, сигнал будет отправлен только этому процессу, но если указать отрицательный номер, сигнал будет отправлен каждому процессу в группе с номером, равным соответствующему положительному числу, т. е. номеру процесса лидера группы процессов. (Удобно для лидера группы процессов то обстоятельство, что идентификатор группы процессов сохраняется в переменной $$.)

Предположим, что программа хочет послать сигнал отбоя (hang-up) всем порожденным процессам, которые она запустила непосредственно, плюс внукам, запущенным этими порожденными процессами, плюс правнукам, запущенным этими внуками, и т. д. Для этого программа сначала должна вызвать setpgrp(0,0), чтобы стать лидером новой группы процессов, и все процессы, которые она создаст, будут входить в новую группу. Не имеет значения, как эти процессы будут запускаться: вручную через fork, автоматически через конвейеризованные open или как фоновые задания посредством system("cmd &"). Даже если у тех процессов будут собственные потомки, отправка сигнала отбоя всей группе процессов найдет их все (за исключением процессов, которые организовали собственную группу или поменяли свой UID, чтобы получить «дипломатический иммунитет» от ваших сигналов).

```
{
    local $SIG{HUP} = "IGNORE";    # исключить себя самого
```

```
    kill(HUP, -$$);                 # послать сигнал своей собственной группе процессов
}
```

Сигнал с номером 0[1] также представляет особый интерес. Этот сигнал не оказывает влияния на целевой процесс, а проверяет, жив ли тот и не поменял ли свой UID. Таким образом, нулевой сигнал проверяет, возможна ли передача сигнала, но фактически не передает его.

```
unless (kill 0 => $kid_pid) {
    warn "что-то нехорошее произошло с $kid_pid";
}
```

Сигнал с номером 0 является единственным сигналом, который работает одинаково в Perl для операционных систем Microsoft и UNIX. В системах Microsoft функция kill фактически не передает сигнал. Вместо этого она заставляет целевой процесс завершиться со статусом, соответствующим номеру сигнала. Когда-нибудь это может быть исправлено. Однако волшебный сигнал 0 работает стандартным неразрушающим образом.

Уборка зомби

По завершении процесса ядро операционной системы посылает его родителю сигнал CHLD, и завершенный процесс превращается в зомби,[2] пока его родитель не вызовет wait или waitpid. Если в Perl запустить другой процесс без помощи fork, всю заботу об уборке зомбированных потомков Perl возьмет на себя, но в случае применения голого fork предполагается, что вы сами убираете за собой. Во многих, но не во всех ядрах простым приемом для автоматической уборки зомби является установка $SIG{CHLD} в "IGNORE". Более гибкий (но трудоемкий) подход состоит в самостоятельной их уборке. В связи с тем что несколько потомков могут умереть, прежде чем вы соберетесь заняться ими, необходимо убирать своих зомби в цикле до тех пор, пока их больше не останется:

```
use POSIX ":sys_wait_h";
sub REAPER { 1 until waitpid(-1, WNOHANG) == -1) }
```

Чтобы запускать этот код должным образом, можно установить обработчик сигнала CHLD:

```
$SIG{CHLD} = \&REAPER;
```

или, если код работает в цикле, сделать так, чтобы «уборщик» запускался на определенных итерациях.

Завершение медленных операций по тайм-ауту

Обычным применением сигналов является ограничение времени выполнения длительных операций. В UNIX (или любой другой POSIX-совместимой системе, поддерживающей сигнал ALRM) можно попросить ядро спустя некоторый промежуток времени отправить процессу сигнал ALRM:

---

[1] Ранее этот сигнал имел символическое имя SIGNULL, но, поскольку его номер однозначно соответствует имени, оно впоследствии перестало поддерживаться, и теперь данный сигнал можно использовать только по его номеру. – *Прим. перев.*

[2] Это действительно технический термин.

```
use Fcntl ":flock";
eval {
    local $SIG{ALRM} = sub { die "перезапуск будильника" };
    alarm 10;              # подать сигнал через 10 секунд
    eval {
        flock(FH, LOCK_EX)  # монопольная блокировка
            || die "невозможно получить блокировку flock: $!";
    };
    alarm 0;               # отменить сигнал
};
alarm 0;                   # защита от ситуации гонки за ресурсами
die if $@ && $@ !~ /перезапуск будильника/; # повторное возбуждение
```

Если аварийный сигнал поступит во время ожидания снятия блокировки, и вы просто перехватите сигнал и выполните возврат, то попадете обратно в функцию flock, потому что Perl автоматически перезапускает системные вызовы, когда это возможно. Единственный выход состоит в том, чтобы возбудить исключительную ситуацию через die, а затем позволить eval перехватить ее. (Это срабатывает, поскольку исключительная ситуация приводит к вызову функции *longjmp*(3) из библиотеки C, которая действительно выводит нас из перезапущенного системного вызова.)

Вложенная ловушка для исключительной ситуации организована здесь потому, что функция flock возбуждает исключительную ситуацию, если на данной платформе системный вызов flock не реализован, а сбросить аварийный сигнал нужно в любом случае. Второй alarm 0 дается на случай, если сигнал придет после выполнения flock, но до того как будет достигнут первый alarm 0. Без второго alarm есть небольшая вероятность попасть в ситуацию гонки за ресурсами, однако величина вероятности не играет роли: ситуация либо существует, либо нет. Мы предпочитаем, чтобы она не существовала.

Блокировка сигналов

Иногда желательно отложить прием сигнала на время выполнения критического фрагмента кода. Просто проигнорировать сигнал нежелательно, но текущая задача слишком важна, чтобы ее можно было прервать. В хеше %SIG блокировка сигналов не реализована, но она осуществляется в модуле POSIX через интерфейс к системному вызову *sigprocmask*(2):

```
use POSIX qw(:signal_h);
$sigset   = POSIX::SigSet->new;
$blockset = POSIX::SigSet->new(SIGINT, SIGQUIT, SIGCHLD);
sigprocmask(SIG_BLOCK, $blockset, $sigset)
    || die "Невозможно заблокировать сигналы INT,QUIT,CHLD: $!\n";
```

Пока эти три сигнала заблокированы, можете делать что угодно, не опасаясь, что вас потревожат. Завершив критический фрагмент, разблокируйте сигналы, восстановив прежнюю маску сигналов:

```
sigprocmask(SIG_SETMASK, $sigset)
    || die "Невозможно восстановить сигналы INT,QUIT,CHLD: $!\n";
```

Если за время действия блокировки поступили какие-либо из этих трех сигналов, они немедленно будут доставлены. Если число необработанных различных

сигналов больше одного, они поступят в произвольном порядке. Кроме того, нельзя определить, какой сигнал сколько раз был получен за время блокировки.[1] Например, если девять порожденных процессов завершились, пока вы блокировали сигналы CHLD, ваш обработчик (если он есть) после разблокирования будет вызван только один раз. Вот поэтому, удаляя зомби, нужно повторять цикл, пока их совсем не останется.

Безопасность сигналов

До версии v5.8 Perl пытался интерпретировать сигналы как прерывания и обрабатывать их незамедлительно, невзирая на состояние интерпретатора. Такой подход был изначально ненадежным из-за проблем реентерабельности. Он мог приводить к повреждению собственной памяти интерпретатора Perl и аварийному завершению процесса – и это еще не самый печальный сценарий.

В настоящее время, когда процессу передается сигнал, Perl просто помечает его как ожидающий обработки. Затем, в более безопасный момент цикла интерпретации, приступает к обработке всех ожидающих сигналов. Это и безопаснее, и аккуратнее, и надежнее, хотя и не всегда своевременно. Некоторые операции, выполняемые Perl, такие как сортировка чрезвычайно большого списка, могут занимать продолжительное время.

Чтобы вернуть Perl к прежнему (неправильному и ненадежному) способу обработки сигналов, назначьте переменной среды PERL_SIGNALS значение "unsafe". Но прежде внимательно ознакомьтесь с разделом «Deferred Signals» (отложенные сигналы) страницы *perlipc* справочного руководства.

Файлы

Возможно, вы никогда ранее не рассматривали файлы как механизм IPC, но на них приходится львиная доля межпроцессных взаимодействий, т.е. значительно бóльшая, чем на все остальные средства, вместе взятые. Когда один процесс помещает свои драгоценные данные в файл, а другой извлекает эти данные, между этими процессами происходит обмен информацией. Среди рассмотренных здесь форм IPC файлы стоят особняком: как свиток папируса, погребенный в пустыне и раскопанный через тысячелетия, файл может быть загружен и прочитан через длительное время после кончины его автора.[2] Популярность файлов не вызывает удивления, если учесть продолжительность хранения и относительную простоту применения.

Использование файлов для передачи информации из далекого прошлого в неизведанное будущее преподносит мало сюрпризов. Файл записывается на какой-то постоянный носитель, например диск, и все. (Если файл содержит HTML, можно сообщить веб-серверу, где его искать.) Интересная задача возникает, если все стороны еще живы и пытаются общаться между собой. В отсутствие некоторого соглашения относительно того, чья очередь сказать свое слово, надежная связь не-

1 Обычно. Подсчет сигналов может быть реализован в некоторых системах реального времени согласно последним спецификациям, но мы таких пока не встречали.

2 Если предположить, что у процесса может быть кончина.

возможна; соглашения можно достичь с помощью блокировки файла, о чем рассказывается в следующем разделе. А через один раздел мы обсудим особые отношения, существующие между родительским процессом и его потомками и позволяющие связанным родством сторонам обмениваться информацией посредством наследования доступа к одним и тем же файлам.

Конечно, возможности файлов ограничены, если речь идет о таких вещах, как удаленный доступ, синхронизация, надежность и управление сеансом. В других разделах данной главы рассказывается о различных механизмах IPC, созданных для преодоления таких ограничений.

Блокировка файлов

В многозадачной среде следует проявлять осторожность, чтобы не вступить в конфликт с другими процессами, которые пытаются обратиться к используемому вами файлу. Если все процессы осуществляют только чтение, проблем не возникает, но как только хотя бы одному процессу понадобится что-нибудь записать в файл, начинается полный хаос, если только какой-нибудь механизм блокировки не выступает в качестве регулировщика движения.

Никогда не используйте существование файла (т. е. проверку `-e $file`) в качестве индикатора блокировки, потому что условие гонки может возникнуть между проверкой существования файла с данным именем и операцией, которую вы собираетесь с ним совершить (например, создать его, открыть или уничтожить). Дополнительно об этом читайте в разделе главы 20 «Разрешение ситуации гонок».

Переносимым интерфейсом Perl для блокировки является функция `flock(`*`HANDLE`*`, `*`FLAGS`*`)`, описываемая в главе 27. Perl добивается максимальной переносимости путем использования самых простых и наиболее распространенных возможностей блокировки, имеющихся на возможно большем числе платформ. Эта семантика достаточно проста для эмуляции в большинстве систем, в том числе таких, которые не поддерживают обычный системный вызов `flock`, например System V или Windows NT. (Если у вас система Microsoft более ранняя, чем NT, то вам, вероятно, не повезло, как и в случае, если у вас система Apple более ранняя, чем OS X.)

Есть две разновидности блокировок: блокировки с совместным (shared) доступом (флаг `LOCK_SH`) и блокировки с монопольным, или взаимоисключающим (exclusive), доступом (флаг `LOCK_EX`). Несмотря на коннотации слова «монопольный», процессы не обязаны подчиняться блокировкам файлов. Смысл в том, что `flock` реализует только *рекомендательную блокировку*, которая не мешает другим процессам читать из файла или писать в него. Запрос монопольной блокировки – это способ, которым процесс позволяет операционной системе приостановить его выполнение, пока все текущие блокировки, будь они с совместным доступом или монопольные, не будут сняты. Аналогично, когда процесс запрашивает блокировку с совместным доступом, он просто приостанавливает свое выполнение, пока не будут сняты все монопольные блокировки. Безопасный совместный доступ к файлу возможен только в случае применения механизма блокировки файлов всеми сторонами.

Поэтому вызов `flock` по умолчанию является блокирующей операцией. Это означает, что, если требуемую блокировку нельзя получить немедленно, операционная система приостановит процесс до тех пор, пока блокировка не станет возможной. Вот как получить блокировку с совместным доступом, применяемую обычно для чтения файла:

```
use Fcntl qw(:DEFAULT :flock);
open(FH, "< filename") || die "невозможно открыть файл с именем: $!";
flock(FH, LOCK_SH)     || die "невозможно блокировать файл с именем: $!"
# теперь выполнить чтение из FH
```

Можно попытаться получить блокировку в неблокирующем режиме, включив флаг `LOCK_NB` в запрос `flock`. Если получить блокировку немедленно не удается, вызов функции тут же завершается неудачей, и она возвращает ложное значение. Например:

```
flock(FH, LOCK_SH | LOCK_NB)
    || die "невозможно блокировать файл: $!";
```

Можно попытаться сделать что-то другое, не возбуждая исключительную ситуацию, как в этом примере, но не вздумайте после неудачного завершения `flock` производить с файлом операции ввода/вывода. Если в блокировке отказано, то, не получив ее, не следует обращаться к файлу. Неизвестно, в каком искаженном состоянии вы получите файл. Основная цель неблокирующего режима в том, чтобы дать возможность «отойти в сторону» и заняться другими делами в период ожидания. Но он также позволяет организовать более дружественное взаимодействие с пользователями, предупреждая их, что для получения блокировки может понадобиться время, чтобы они не чувствовали себя брошенными на произвол судьбы:

```
use Fcntl qw(:DEFAULT :flock);
open(FH, "< filename") || die " невозможно открыть файл с именем  $!";
unless (flock(FH, LOCK_SH | LOCK_NB)) {
    local $| = 1;
    print "Ждем блокировки файла.. ",
    flock(FH, LOCK_SH) || die " невозможно заблокировать файл с именем: $!
    print "блокировка получена.\n"
}
# теперь можно читать из FH
```

Некоторые разработчики испытают соблазн поместить такую неблокирующую блокировку в цикл. Основная проблема с неблокирующим режимом в том, что к моменту, когда вы снова выполните проверку, может оказаться, что кто-то другой уже заблокировал файл, потому что вы отошли от своего места в очереди. Иногда приходится просто стоять и ждать. Если повезет, под рукой окажутся журналы, которые можно почитать.

Блокируются дескрипторы файлов, а не их имена.[1] При закрытии файла блокировка автоматически снимается, будь оно произведено явно, вызовом `close`, или косвенно, путем повторного открытия указателя или завершения процесса.

[1] Фактически блокируются не дескрипторы файлов (filehandles), а указатели файлов уровня операционной системы, связанные с дескрипторами, поскольку операционная система ничего не знает о дескрипторах файлов Perl. Это означает, что все сообщения `die` о невозможности получить блокировку по имени файла технически являются неточными. Но сообщения об ошибках вида «Не могу получить блокировку файла, представленного указателем файла, связанным с дескриптором файла, первоначально открытым для файла с именем *filename*, хотя в данный момент имя *filename* может представлять совершенно другой файл, нежели наш дескриптор» просто собьют с толку пользователя (не говоря уже о читателе).

При получении монопольной блокировки, обычно для записи, нужно проявлять еще бо́льшую осторожность. Для этого не годится обычная функция open; в случае применения режима открытия < возникает ошибка, если файл еще не существует, а установив режим >, можно разрушить существующий файл. Чтобы заблокировать файл перед записью в него, используйте sysopen. Успешно открыв файл для записи и еще не тронув его, нужно получить монопольную блокировку и *только тогда* усекать файл. После этого можно записать в него новые данные.

```
use Fcntl qw(:DEFAULT :flock);
sysopen(FH, "filename", O_WRONLY | O_CREAT)
    || die "невозможно открыть файл с именем: $!";
flock(FH, LOCK_EX)
    || die "невозможно заблокировать файл с именем: $!"
truncate(FH, 0)
    || die "невозможно сделать усечение файла с именем: $!"
# теперь можно писать в FH
```

Если потребуется изменить некоторую часть файла, опять-таки используйте sysopen. На этот раз следует запросить доступ для чтения и записи одновременно, при необходимости создав файл. Когда файл открыт, прежде чем выполнять чтение или запись, нужно получить монопольную блокировку и сохранять ее на все время операции. Снять блокировку лучше всего путем закрытия файла, так как это гарантирует запись всех буферов на диск перед снятием блокировки.

При обновлении производится чтение старых значений и запись новых. Обе операции нужно выполнять во время действия одной монопольной блокировки, чтобы другой процесс не прочел (неизбежно неверные) данные после (или даже прежде) того, как это сделаете вы, но перед тем как вы закончите запись. (Мы вернемся к этой ситуации при рассказе о совместно используемой памяти далее в этой главе.)

```
use Fcntl qw(:DEFAULT :flock);

sysopen(FH, "counterfile", O_RDWR | O_CREAT)
    || die "невозможно открыть counterfile: $!"
flock(FH, LOCK_EX)
    || die "невозможно заблокировать counterfile для записи $
$counter = <FH> || 0; # в первый раз будет undef
seek(FH, 0, 0)
    || die "невозможно вернуться в начало counterfile : $!
print FH $counter+1, "\n"
    || die "невозможно выполнить запись в counterfile: $!

# следующая строка в этой программе технически избыточна, но
# в общем случае мысль правильная
truncate(FH, tell(FH))
    || die " невозможно выполнить усечение counterfile: $!";
close(FH)
    || die " невозможно закрыть counterfile $!";
```

Нельзя заблокировать неоткрытый файл и нельзя применить одну блокировку к нескольким файлам. Однако можно использовать отдельный файл в качестве «семафора», чтобы обеспечить управление доступом к чему-то еще посредством обычных блокировок этого файла-«семафора» в режиме совместного или моно-

польного доступа. У такого подхода есть несколько преимуществ. Можно иметь один файл блокировки, управляющий доступом к нескольким файлам, избегая при этом такого взаимного блокирования (deadlock), которое происходит, когда один процесс пытается заблокировать файлы в одном порядке, а другой – те же файлы в другом порядке. Файл-«семафор» можно использовать для блокировки целого каталога с файлами. Можно даже управлять доступом к тому, что не находится в файловой системе, например к объекту совместного доступа в памяти или сокету, к которому несколько серверов, полученных путем fork-ветвления, намерены применить вызов accept.

Если у вас есть файл DBM, не предоставляющий собственного механизма блокировки, одновременным доступом к нему нескольких агентов лучше всего управлять с использованием вспомогательного файла блокировки. В противном случае внутренний кэш библиотеки DBM может рассинхронизироваться с дисковым файлом. Прежде чем вызывать dbmopen или tie, откройте и заблокируйте файл-«семафор». Если вы открываете базу данных с флагом O_RDONLY, для блокировки потребуется установить флаг LOCK_SH. В противном случае используйте LOCK_EX для монопольного доступа с целью обновления базы данных. (Это действенно, только если все участники согласны обращать внимание на «семафор».)

```
use Fcntl qw(:DEFAULT :flock);
use DB_File; # только в целях демонстрации; годится любая БД

$DBNAME = "/path/to/database";
$LCK = $DBNAME . ".lockfile";

# используйте O_RDWR, если собираетесь что-то поместить в файл блокировки
sysopen(DBLOCK, $LCK, O_RDONLY | O_CREAT)
    || die "невозможно открыть $LCK: $!";

# нужно получить блокировку перед открытием базы данных
flock(DBLOCK, LOCK_SH)
    || die "невозможно LOCK_SH $LCK: $!";

tie(%hash, "DB_File", $DBNAME, O_RDWR | O_CREAT)
    || die "невозможно tie $DBNAME: $!";
```

Теперь можно без опаски совершать любые действия с хешем %hash, связанным с базой данных. Закончив работу с базой данных, обеспечьте явное закрытие этих ресурсов, причем в порядке, обратном порядку их получения:

```
untie %hash;  # закрыть базу данных прежде файла блокировки
close DBLOCK; # теперь безопасно снять блокировку
```

Если у вас установлена библиотека GNU DBM, можно использовать неявную блокировку из стандартного модуля GDBM_File. Если только начальный вызов tie не содержит флага GDBM_NOLOCK, библиотека разрешает одновременно открывать файл GDBM для записи только одному процессу и запрещает одновременно открывать базу данных записывающим и читающим процессам.

Передача дескрипторов файлов

При создании порожденного процесса с помощью fork новый процесс наследует все открытые дескрипторы файлов родителя. Использование дескрипторов файлов

для взаимодействия между процессами легче всего показать на обычных файлах. Важно понять, как это работает, чтобы овладеть более сложными механизмами на основе конвейеров и сокетов, описываемыми далее в этой главе.

Следующий простой пример открывает файл и запускает порожденный процесс. Затем порожденный процесс использует дескриптор файла, уже открытый для него:

```
open(INPUT, "< /etc/motd") || die "/etc/motd: $!";
if ($pid = fork) { waitpid($pid,0) }
else {
    defined($pid)          || die "fork: $!";
    while (<INPUT>) { print "$.: $_" }
    exit; # не дать порожденному процессу вернуться в основной код
}
# Теперь указатель INPUT в родителе находится в EOF
```

Доступ к файлу, открытому вызовом open, сохраняется до закрытия дескриптора файла; изменение прав доступа к файлу или привилегий владельца файла не оказывает влияния на доступ. Даже если процесс позже изменит своего владельца (пользователя или группу) либо изменится принадлежность файла, это не окажет влияния на уже открытые дескрипторы файлов. Программы, выполняющиеся с повышенными привилегиями (такие, как системные демоны или программы, запускаемые от имени привилегированного пользователя), часто открывают файл, а затем передают указатель порожденному процессу, который не смог бы открыть его самостоятельно.

Хотя эта возможность очень удобна в случае ее целевого применения, она может также создавать проблемы с защитой данных при непреднамеренной утечке дескрипторов файлов из одной программы в другую. Чтобы избежать неявной передачи доступа к любым указателям файлов, Perl автоматически закрывает любые открытые им дескрипторы файлов (в том числе конвейеры и сокеты) при явном запуске новой программы через exec или неявном выполнении ее через вызов конвейеризованного open, system или qx// (обратные апострофы). Системные дескрипторы файлов STDIN, STDOUT и STDERR являются исключением, поскольку их основное назначение – обеспечить связь между программами. Поэтому одним из способов передачи дескриптора файла новой программе является его копирование в один из стандартных дескрипторов:

```
open(INPUT, "< /etc/motd") || die "/etc/motd: $!";
if ($pid = fork) { wait }
else {
    defined($pid)          || die "fork: $!";
    open(STDIN, "<&INPUT") || die "dup: $!";
    exec("cat", "-n")      || die "exec cat: $!";
}
```

Если и впрямь требуется, чтобы новая программа получила доступ к указателю файла, отличному от этих трех, это возможно, но вам придется проделать одно из двух. Когда Perl открывает новый файл (конвейер или сокет), он проверяет текущее значение переменной $^F ($SYSTEM_FD_MAX). Если числовой указатель файла, используемый новым дескриптором файла, больше $^F, указатель помечается как подлежащий закрытию. В противном случае Perl оставит его в покое, и новые программы, которые вы запустите, унаследуют доступ.

Не всегда легко предсказать, какой числовой идентификатор будет иметь вновь открываемый дескриптор файла, но можно временно установить максимальный указатель файла в системе равным какому-нибудь очень большому числу:

```
# открыть файл и пометить INPUT как открытый на время выполнения exec
{
    local $^F = 10_000;
    open(INPUT, "< /etc/motd") || die "/etc/motd: $!";
} # прежнее значение $^F восстанавливается при выходе из области видимости
```

Теперь необходимо заставить новую программу обратить внимание на номер указателя только что открытого дескриптора файла. Самое правильное (для систем, которые это поддерживают) – передать специальное имя файла, соответствующее указателю файла. Если в системе есть каталог с именем */dev/fd* или */proc/$$/fd*, содержащий файлы с номерами от 0 до максимального числа поддерживаемых указателей, вам, возможно, удастся воспользоваться такой стратегией. (Во многих системах Linux есть оба каталога, но только */proc* содержит то, что нужно. В BSD и Solaris это каталог */dev/fd*. Вам придется выяснить, какой из каталогов вашей операционной системы лучше подходит для ваших нужд.) Сначала откройте и пометьте дескриптор файла как открытый для exec, как показано в приведенном выше фрагменте кода, затем выполните ветвление, как показано ниже:

```
if ($pid = fork) { wait }
else {
    defined($pid)              || die "fork: $!";
    $fdfile = "/dev/fd/" . fileno(INPUT);
    exec("cat", "-n", $fdfile) || die "exec cat: $!";
}
```

Если в системе поддерживается системный вызов fcntl, можно вручную изменить флаг закрытия дескриптора файла при выполнении exec. Это удобно, когда при создании дескриптора файла не было известно, что его потребуется использовать совместно с порожденными процессами.

```
use Fcntl qw/F_SETFD/;

fcntl(INPUT, F_SETFD, 0)
    || die "Не могу сбросить флаг закрытия при exec для INPUT: $!\n";
```

Можно также принудительно закрыть дескриптор файла:

```
fcntl(INPUT, F_SETFD, 1)
    || die "Не могу установить флаг закрытия при exec для INPUT: $!\n";
```

А можно запросить текущее состояние:

```
use Fcntl qw/F_SETFD F_GETFD/;

printf("INPUT будет %s для exec\n",
    fcntl(INPUT, F_GETFD, 1) ? "закрыт" : "оставлен открытым");
```

Если система не поддерживает хранение числовых указателей файлов в файловой системе, и нужно передать дескриптор файла, отличный от STDIN, STDOUT или STDERR, эту проблему можно решить, но придется выполнить дополнительные телодвижения. Обычно данная задача решается передачей номера числового указателя через переменную среды или параметр командной строки.

Если выполняемая программа написана на Perl, преобразовать числовой указатель файла в дескриптор можно посредством open. Вместо имени файла используйте номер указателя с приставкой &=.

```
if (($ENV{input_fdno} // "") =~ /^\d$/) {
    open(INPUT, "<&=$ENV{input_fdno}")
        || die "невозможно fdopen $ENV{input_fdno} для ввода: $!";
}
```

Все становится проще, если нужно запустить подпрограмму Perl или программу, ожидающую получить имя файла в качестве аргумента. Автоматически открыть файл по системному указателю можно посредством обычной функции open (но не sysopen или open с тремя аргументами). Представьте, что имеется такая несложная программа на Perl:

```
#!/usr/bin/perl -p
# nl - число строк ввода
printf "%6d ", $.;
```

Если предположить, что указатель INPUT настроен так, чтобы он оставался открытым при выполнении exec, эту программу можно вызывать следующим образом:

```
$fdspec = "<&=" . fileno(INPUT);
system("nl", $fdspec);
```

или перехватить вывод:

```
@lines = `nl '$fdspec' `; # одинарные кавычки защищают spec от оболочки
```

Запускаете вы другую программу или нет, при использовании числовых указателей файлов, наследуемых через fork, существует одна маленькая ловушка. В отличие от переменных, которые копируются вызовом fork и становятся идентичными, но независимыми копиями, дескрипторы файлов в обоих процессах действительно будут одни и те же. Если какой-то процесс читает данные из дескриптора, позиция в файле в другом процессе также увеличивается, и эти данные становятся недоступными в обоих процессах. Если процессы выполняют чтение поочередно, возникает своеобразная чехарда. Это очевидно в случае дескрипторов, работающих с устройствами последовательного доступа, конвейерами и сокетами, поскольку такие устройства обычно доступны только для чтения, и данные в них хранятся лишь до первой операции чтения, а вот подобное поведение дисковых файлов может оказаться неожиданным. Если в результате возникают проблемы, после вызова fork откройте заново файлы, требующие раздельного просмотра.

Оператор fork является наследием UNIX, и это значит, что он может быть некорректно реализован на платформах, отличных от UNIX/POSIX. Примечательно, что fork работает в системах Microsoft только для Perl 5.6 (или выше) в Windows 98 (или выше). В этих системах fork реализуется посредством запуска нескольких потоков выполнения внутри одной программы, но это не те потоки, которые по умолчанию совместно используют все данные; в данном случае речь только о дескрипторах файлов.

Каналы

Канал (*pipe*) – это однонаправленный вектор ввода/вывода, способный передавать поток байтов от одного процесса к другому. Каналы могут быть как имено-

ванными, так и безымянными. Вам, вероятно, лучше знакомы безымянные каналы, поэтому с них и начнем.

Анонимные каналы

Функция open откроет канал, а не файл, если перед вторым аргументом или после него поставить символ конвейера. В результате оставшиеся аргументы превращаются в команду, которая будет интерпретирована как процесс (или группа процессов), принимающий или передающий поток данных. Ниже приведен способ запуска порожденного процесса, в который осуществляется запись:

```
open SPOOLER, "| cat -v | lpr -h 2>/dev/null"
    || die "can't fork: $!";
local $SIG{PIPE} = sub { die "spooler pipe broke" };
print SPOOLER "stuff\n";
close SPOOLER or die "bad spool: $! $?";
```

Фактически мы запускаем здесь два процесса. Вывод в первый процесс (*cat*) осуществляется напрямую. Второй процесс (*lpr*) получает выходной поток первого процесса. В разработке сценариев для интерпретатора команд это часто называется *конвейером* (*pipeline*). Конвейер может объединять в цепочку любое количество процессов, при условии, что все промежуточные процессы умеют работать в режиме *фильтра*, т. е. читать из стандартного ввода и писать в стандартный вывод.

Perl использует системный интерпретатор команд по умолчанию (*/bin/sh* в UNIX), если команда конвейера содержит специальные символы интерпретатора команд. Если же запускается только одна команда, и нет необходимости обращаться к интерпретатору команд, можно использовать формат канала с несколькими аргументами:

```
open SPOOLER, "|-", "lpr", "-h" # требует 5.6.1
    || die "can't run lpr: $!";
```

Если повторно открыть стандартный поток вывода программы как канал в другую программу, то с этого момента весь вывод в STDOUT будет поступать на стандартный ввод новой программы. Поэтому для организации постраничного вывода[1] следует использовать:

```
if (-t STDOUT) {               # только если stdout является терминалом
    my $pager = $ENV{PAGER} || "more";
    open(STDOUT, "| $pager")   || die "невозможно запустить пейджер: $!";
}
END {
    close(STDOUT)              || die "невозможно закрыть STDOUT: $!"
}
```

После записи в дескриптор файла, ассоциированный с каналом, и окончания работы с этим дескриптором всегда явно закрывайте его функцией close. Благодаря этому основная программа не завершит работу раньше своих потомков.

Вот как запустить порожденный процесс, из которого предполагается осуществлять чтение:

---

[1] То есть чтобы просматривать по одному экрану за раз.

```
open STATUS, "netstat -an 2>/dev/null |"
    || die "can't fork: $!";
while (<STATUS>) {
    next if /^(tcp|udp)/;
    print;
}
close STATUS || die "bad netstat: $! $?";
```

Открыть многоступенчатый канал для ввода можно так же, как для вывода. И, как и раньше, можно обойти интерпретатор команд, используя альтернативную форму open:

```
open STATUS, "-|", "netstat", "-an" # требует 5.6.1
    || die "can't run netstat: $!";
```

Но при этом вы не получите переадресации ввода/вывода, раскрытия по маске и многоступенчатых конвейеров, поскольку Perl полагается в этом на интерпретатор команд.

Возможно, вы заметили, что обратные апострофы дают тот же эффект, что открытие канала для чтения:

```
print grep { !/^(tcp|udp)/ } `netstat -an 2>&1`;
die "bad netstat" if $?;
```

Обратные апострофы чрезвычайно удобны, но данная операция загружает в память сразу весь объем данных, поэтому часто оказывается более эффективно открыть дескриптор файла для канала и обрабатывать данные построчно. Так мы получаем более тонкий контроль над всей операцией, имея возможность досрочно остановить порожденный процесс, если это потребуется. Большей эффективности можно также добиться, обрабатывая входные данные по мере поступления, поскольку компьютеры способны чередовать разные операции при одновременном выполнении двух или более процессов. (Даже на машине с одним центральным процессором операции ввода и вывода могут происходить, когда процессор занят чем-то другим.)

Из-за одновременного выполнения двух или более процессов катастрофа может постичь порожденный процесс в любой момент между вызовами open и close. Это означает, что родитель должен проверять значения, возвращаемые функциями open и close. Проверки значения одной только open недостаточно, поскольку она может сообщить лишь об успехе fork и, возможно, о том, что следующая команда была успешно запущена. (Да и такую информацию можно получить только в новых версиях Perl и только если команда выполнена непосредственно ответвившимся потомком, а не через интерпретатор команд.) Обо всех катастрофах, происходящих после этого, потомок сообщает родителю ненулевыми кодами завершения. Когда функция close получает такой код, она возвращает ложное значение, указывая, что фактическое значение состояния хранится в переменной $? ($CHILD_ERROR). Поэтому проверка значения, возвращаемого close, столь же важна, как проверка open. Осуществляя запись в канал, следует приготовиться к обработке сигнала PIPE, который посылается, если процесс на другом конце завершился раньше, чем передача ему данных.

Разговор с самим собой

Другой подход к IPC состоит в том, чтобы заставить программу, так сказать, разговаривать с самой собой. В действительности процесс общается через каналы с собственной копией. Этот механизм действует во многом как открытие канала, о котором мы говорили в предыдущем разделе, за исключением того, что порожденный процесс продолжает выполнять тот же сценарий, а не какую-либо другую команду.

Чтобы реализовать это с помощью функции open, используется псевдокоманда, состоящая из знака «минус». Поэтому второй аргумент open выглядит как "-|" или "|-" в зависимости от того, хотите вы развернуть конвейер от себя или к себе. Как и обычная команда fork, функция open возвращает идентификатор порожденного процесса в родительский процесс и 0 – в порожденный процесс. Другое отличие состоит в том, что дескриптор файла, объявленный в open, используется только в родительском процессе. В порожденном процессе конец канала прикрепляется к STDIN или STDOUT. То есть, если канал открывается *к* минусу ("|-"), в него можно вывести данные, которые потомок найдет в STDIN:

```
if (open(TO, "|-")) {
    print TO $fromparent;
}
else {
    $tochild = <STDIN>;
    exit;
}
```

Если канал открывается *от* минуса ("-|"), из него можно читать данные, которые потомок запишет в STDOUT:

```
if (open(FROM, "-|")) {
    $toparent = <FROM>;
}
else {
    print STDOUT $fromchild;
    exit;
}
```

Обычно эта конструкция используется для обхода интерпретатора команд, когда необходимо открыть канал внутри команды. Это может потребоваться, например, чтобы избежать интерпретации содержащихся в команде файловых масок. Версия Perl 5.6.1 (или более поздняя) позволяет получить тот же результат, используя формат open с несколькими аргументами.

Версию open, выполняющую ветвление, можно также использовать, чтобы безопасно открыть файл или команду, даже если вы работаете под необходимым UID или GID. Создаваемый функцией fork потомок отбрасывает любые дополнительные права доступа, затем безопасно открывает файл или команду и действует как посредник, передавая данные между своим более мощным родителем и открытым файлом или командой. Примеры можно найти в разделе «Доступ к командам и файлам при наличии ограниченных прав» главы 20.

Версию open, выполняющую fork-ветвление процессов, можно творчески применять для фильтрации собственного вывода программы. Некоторые алгоритмы значительно проще реализовать в два отдельных прохода, чем за один. Вот простой пример, эмулирующий программу UNIX *tee*(1) путем перенаправления обыч-

ного вывода в конвейер. Агент на другом конце конвейера (одна из наших собственных подпрограмм) распределяет вывод во все указанные файлы:

```
tee("/tmp/foo", "/tmp/bar", "/tmp/glarch");

while (<>) {
    print "$ARGV at line $. => $_";
}
close(STDOUT) || die "невозможно закрыть STDOUT: $!";

sub tee {
    my @output = @_;
    my @handles = ();
    for my $path (@output) {
        my $fh; # будет инициализирована вызовом open ниже
        unless (open ($fh, ">", $path)) {
            warn "невозможно записать в $path: $!";
            next;
        }
        push @handles, $fh;
    }

    # повторно открыть STDOUT в родителе и вернуться
    return if my $pid = open(STDOUT, "|-");
    die "невозможно выполнить ветвление: $!" unless defined $pid;

    # обработать STDIN в порожденном процессе
    while (<STDIN>) {
        for my $fh (@handles) {
            print $fh $_ || die "отказ при выводе из tee: $!";
        }
    }
    for my $fh (@handles) {
        close($fh) || die "отказ при закрытии tee: $!";
    }
    exit; # не позволяйте потомку возвращаться в основную программу!
}
```

Этот прием можно применять многократно, чтобы поместить в поток вывода необходимое количество фильтров. Просто продолжайте вызывать функции, открывающие STDOUT **с ветвлением процессов, и пусть порожденный процесс читает из родительского (который он видит как** STDIN**) и передает обработанный вывод далее, следующей функции в потоке.**

Другим интересным применением «разговора с самим собой» является перехват вывода «плохо воспитанной» функции, которая постоянно выводит свои результаты в STDOUT**. Представьте, что в Perl есть только функция** printf**, а** sprintf **нет. Тогда потребуется что-то, работающее как обратные апострофы, но с функциями Perl, а не с внешними командами:**

```
badfunc("arg");                        # черт, убежала!
$string = forksub(\&badfunc, "arg"); # перехватить как строку
@lines  = forksub(\&badfunc, "arg"); # как отдельные строки
sub forksub {
    my $kidpid = open my $self, "-|";
```

```
        defined $kidpid         || die "невозможно ветвление: $!",
        shift->(@_), exit       unless $kidpid;
        local $/                unless wantarray;
        return <$self>;         # закрывается при выходе из области видимости
    }
```

Мы не утверждаем, что этот код эффективен; связанный дескриптор файла работал бы, вероятно, значительно быстрее. Но написать такой код намного проще, когда вы торопитесь сильнее, чем ваш компьютер.

Двунаправленная связь

Использование функции open для объединения с другой командой через канал достаточно хорошо работает при однонаправленной связи, но как организовать двунаправленный обмен? Подход, который кажется очевидным, на самом деле не работает:

```
open(PROG_TO_READ_AND_WRITE, "| некоторая программа |") # НЕВЕРНО!
```

а если вы забыли включить вывод предупреждений, то полностью пропустите следующее диагностическое сообщение:

```
Can't do bidirectional pipe at myprog line 3.
[Невозможно создать двунаправленный канал в myprog строка 3]
```

Функция open не позволяет сделать это, потому что подвержена взаимоблокировкам, если не проявлять большую осторожность. Но если очень хочется, то можно использовать стандартный библиотечный модуль IPC::Open2, чтобы подключить два канала к STDIN и STDOUT подпроцесса. Имеется также модуль IPC::Open3 для тринаправленного ввода/вывода (позволяющий перехватывать и STDERR порожденного процесса), но для него требуется либо «неуклюжий» цикл select, либо более удобный модуль IO::Select. Но тогда придется отказаться от буферизованных операций ввода Perl, таких как <> (readline).

Вот пример использования open2:

```
use IPC::Open2;
local (*Reader, *Writer),
$pid = open2(\*Reader, \*Writer, "bc -l")
$sum = 2;
for (1 .. 5) {
    print Writer "$sum * $sum\n";
    chomp($sum = <Reader>);
}
close Writer;
close Reader;
waitpid($pid, 0);
print "сумма равна $sum\n"
```

Можно также вызывать автовивификацию лексических дескрипторов файлов:

```
my ($fhread, $fhwrite);
$pid = open2($fhread, $fhwrite, "cat -u -n");
```

Проблема здесь в целом связана с тем, что стандартная буферизация ввода/вывода испортит вам настроение на весь день. Хотя буфер дескриптора файла вывода выталкивается автоматически (это делает библиотека) и процесс на другом конце

получает данные своевременно, обычно никакими силами нельзя заставить адресата оказать взаимную любезность. В данном конкретном случае нам повезло: *bc* рассчитывает на работу через канал (pipe) и умеет выталкивать буфер для каждой выводимой строки. Но мало какие команды действуют именно так, поэтому такой способ редко срабатывает, если только не вы сами написали программу, находящуюся на другом конце двунаправленного канала. Даже простые и, очевидно, интерактивные программы, такие как *ftp*, здесь пасуют, поскольку в них не применяется построчная буферизация канала; лишь построчная буферизация устройства tty.

На помощь могут придти модули `IO::Pty` и `Expect` из CPAN, которые предоставляют устройство tty (фактически псевдоустройство tty, но оно действует как настоящее). Благодаря этому мы получаем построчную буферизацию в другом процессе, не модифицируя его.

Если потребуется разбить программу на несколько процессов и для каждого обеспечить возможность двустороннего общения, то из попытки использовать для этого интерфейсы каналов высокого уровня ничего не выйдет, потому что все они – однонаправленные. Потребуется два вызова низкоуровневой функции `pipe`, по одному на каждое направление передачи:

```
pipe(FROM_PARENT, TO_CHILD)     || die "pipe: $!";
pipe(FROM_CHILD,  TO_PARENT)    || die "pipe: $!";
select((select(TO_CHILD), $| = 1))[0]);  # автоматическое выталкивание
select((select(TO_PARENT), $| = 1))[0]); # автоматическое выталкивание

if ($pid = fork) {
    close FROM_PARENT; close TO_PARENT;
    print TO_CHILD "Родительский процесс Pid $$ отправляет это\n";
    chomp($line = <FROM_CHILD>);
    print "Родительский процесс Pid $$ только что прочел:  $line'\n";
    close FROM_CHILD; close TO_CHILD;
    waitpid($pid,0);
} else {
    die "невозможно ветвление: $!" unless defined $pid;
    close FROM_CHILD; close TO_CHILD;
    chomp($line = <FROM_PARENT>);
    print "Порожденный процесс Pid $$ только что прочел: '$line'\n";
    print TO_PARENT "Порожденный процесс Pid $$ отправляет это\n";
    close FROM_PARENT; close TO_PARENT;
    exit;
}
```

Во многих системах UNIX не требуется выполнять два отдельных вызова `pipe`, чтобы установить полнодуплексную связь между родительским и порожденным процессами. Системный вызов `socketpair` обеспечивает двусторонние соединения между связанными процессами на одной и той же машине. Поэтому вместо двух каналов достаточно одной пары сокетов.

```
use Socket;
socketpair(Child, Parent, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    || die "socketpair: $!";

# или позвольте Perl выбрать имена файлов за вас
my ($kidfh, $dadfh);
```

```
socketpair($kidfh, $dadfh, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    || die "socketpair: $!";
```

После вызова fork родительский процесс закрывает указатель Parent и осуществляет чтение и запись через указатель Child. Между тем порожденный процесс закрывает указатель Child, после чего читает и записывает через указатель Parent.

Если двунаправленная связь интересует вас потому, что процесс-адресат реализован как стандартная служба Интернета, то, вероятно, следует отказаться от посредников и применить модуль CPAN, предназначенный специально для этой цели. (Перечень некоторых модулей такого рода приведен в разделе «Сокеты».)

Именованные каналы

Именованный канал (named pipe), часто называемый FIFO[1], является механизмом обмена данными между неродственными процессами на одной и той же машине. Имена «именованных» каналов сохраняются в файловой системе, и это просто необычный способ сказать, что в пространство имен файловой системы можно поместить особый файл, за которым стоит не диск, а некий процесс.[2] FIFO удобно использовать, когда необходимо соединиться с неродственным процессом. Процесс, открывающий FIFO, блокируется до тех пор, пока не появится процесс на другом конце. То есть, если читающий процесс откроет FIFO первым, он заблокируется, пока не появится пишущий процесс, и наоборот.

Для создания именованного канала используйте функцию POSIX mkfifo – разумеется, если ваша система поддерживает стандарт POSIX. Пользователям систем Microsoft придется заглянуть в модуль Win32::Pipe, который, несмотря на название, создает как раз именованные каналы. (Анонимные каналы пользователи Win32 создают, как и все мы, посредством pipe.)

Допустим, например, что каждая операция чтения из файла *.signature* должна возвращать новый результат. Сделайте файл именованным каналом, на другом конце которого сидит программа Perl и «выплевывает» случайным образом выбранные мудрые изречения. Теперь, когда любая программа (почтовая, для чтения телеконференций, finger и т. д.) попытается прочесть этот файл, она подключится к нашей программе и получит динамически созданную подпись.

В следующем примере используется редко встречающийся оператор проверки файла -p, чтобы выяснить, не удален ли кем-нибудь (или чем-нибудь) наш канал FIFO.[3] Если да, то нечего и пытаться открыть его, и мы рассматриваем это как повод для завершения. Вызови мы простую функцию open в режиме "> $fpath", возникла бы ситуация гонки, чреватая риском создания обычного файла *.signature*, если бы он отсутствовал между проверкой -p и вызовом open. Режим "+< $fpath" тоже не годится, потому что открытие FIFO (и только FIFO) для чтения-записи не является блокирующим. Вызвав sysopen без флага O_CREAT, мы обойдем эту проблему, и файл ни при каких обстоятельствах не будет создаваться случайно.

---

[1] First In First Out – первым пришел, первым ушел. – *Прим. перев.*

[2] То же можно делать для сокетов домена UNIX, но с ними нельзя использовать open.

[3] С его помощью можно также проверить связь некоторого дескриптора с каналом, именованным или безымянным, например -p STDIN.

```
use Fcntl,                # для sysopen
chdir;                    # в исходный каталог
$fpath = ".signature";
$ENV{PATH} .= ":/usr/games";

unless (-p $fpath) {   # не канал,
    if (-e _) {           # а что-то иное
        die "$0: не буду перезаписывать .signature\n";
    } else {
        require POSIX;
        POSIX::mkfifo($fpath, 0666) || die "невозможно создать $fpath: $!";
        warn "$0: создан именованный канал $fpath\n";
    }
}

while (1) {
    # выйти, если файл .signature удален вручную
    die "Файл канала исчез" unless -p $fpath;
    # следующая строка заблокирует выполнение, пока не появится читатель
    sysopen(FIFO, $fpath, O_WRONLY)
        || die "невозможно открыть для записи $fpath: $!";
    print FIFO "John Smith (smith\@host.org)\n", `fortune -s`;
    close FIFO;
    select(undef, undef, undef, 0.2); # приостановка на 1/5 секунды
}
```

Небольшая пауза после закрытия нужна, чтобы дать читающему процессу возможность прочесть то, что записано. Если сразу повторить цикл и снова открыть FIFO, прежде чем читающий процесс закончит чтение только что переданных данных, он не увидит признак конца файла, так как в работу снова вступит пишущий процесс. Так мы и будем ходить по кругу, пока на какой-нибудь итерации пишущий процесс немного не отстанет и читающий процесс наконец не увидит этот неуловимый конец файла. (А мы беспокоились о ситуации гонки!)

System V IPC

System V IPC ненавидят все. Этот механизм работает медленнее, чем перфокарты, прирезает для себя коварные маленькие пространства имен, совершенно не связанные с файловой системой, использует для именования своих объектов враждебные человеческому восприятию числа и постоянно теряет ход собственных мыслей. Очень часто перед системным администратором встает задача «найти и уничтожить»: выследить потерянные объекты SysV IPC с помощью *ipcs*(1) и убить их с помощью *ipcrm*(1) – по возможности раньше, чем кончится системная память.

Несмотря на все эти муки, древний IPC SysV все же имеет ряд возможных применений. Перечислим три типа объектов IPC: совместно используемая память, семафоры и сообщения. Для передачи сообщений предпочтительным механизмом сегодня являются сокеты, которые также имеют значительно более высокую переносимость. Для простых семафоров обычно используется файловая система. А что касается совместно используемой памяти, то это решать вам. Если у вас есть в ней потребность, то более современный вызов *mmap*(2) вам поможет, хотя качество его реализации зависит от используемой системы. Требуется также про-

являть некоторую осторожность, чтобы Perl не переместил ваши строки из того места, куда их положит *mmap*(2).

Модуль File::Map из архива CPAN существенно упрощает работу. Он по-прежнему требует некоторой осторожности, но если вы что-то упустите из виду, он просто выведет сообщение, вместо того чтобы свалить ядро с ошибкой нарушения прав доступа к памяти (segmentation violation).

Ниже приводится небольшая программа, демонстрирующая управляемый доступ к буферу совместно используемой памяти целого «выводка» процессов-братьев. Объекты SysV IPC могут совместно использоваться и *неродственными* процессами на одном и том же компьютере, но тогда придется решить, как они будут друг друга определять. Для безопасности мы создадим для каждого по семафору.[1]

Перед каждой операцией чтения или записи в совместно используемую память необходимо сначала пройти семафор. Это может стать довольно утомительным, поэтому создадим для доступа объект-обертку. Модуль IPC::Shareable идет на шаг дальше, оборачивая свой класс объектов в интерфейс tie.

Эта программа выполняется, пока вы не прервете ее с помощью комбинации Control-C или эквивалентной:

```
#!/usr/bin/perl -w
use v5.6.0;            # или выше
use strict;
use sigtrap qw(die INT TERM HUP QUIT);
my $PROGENY = shift(@ARGV) || 3;
eval { main() };       # смотрите ниже в DESTROY, зачем это нужно
die if $@ && $@ !~ /^Caught a SIG/;
print "\nDone.\n";
exit;

sub main {
    my $mem = ShMem->alloc("Original Creation at     localtime);
    my(@kids, $child);
    $SIG{CHLD} = "IGNORE";
    for (my $unborn = $PROGENY; $unborn > 0; $unborn--) {
        if ($child = fork) {
            print "$$ запущен порожденный процесс $child\n";
            next;
        }
        die "невозможно ветвление: $!" unless defined $child;
        eval {
            while (1) {
                $mem->lock();
                $mem->poke("$$    . localtime)
                    unless $mem->peek =~ /^$$\b/o,
                $mem->unlock();
            }
```

[1] Практичнее было бы создать пару семафоров для каждой области памяти совместного доступа – по одному для чтения и записи. Фактически это и делает модуль IPC::Shareable из CPAN. Но мы стараемся сохранить простоту. Стоит, однако, признать, что с помощью пары семафоров можно было бы воспользоваться единственной, пожалуй, привлекательной особенностью SysV IPC – возможностью выполнять атомарные операции над целыми группами семафоров, что иногда полезно.

```
        };
        die if $@ && $@ !~ /^Caught a SIG/;
        exit;  # выход из порожденного процесса
    }
    while (1) {
        print "Содержимое буфера: ", $mem->get, "\n";
        sleep 1;
    }
}
```

А вот листинг пакета ShMem, используемого в программе. Его можно поместить в конец программы или в отдельный файл (с вызовом "1; " в конце) и загрузить из основной программы посредством require. (Два модуля IPC, используемые в нем, присутствуют в стандартном дистрибутиве Perl.)

```
package ShMem;
use IPC::SysV qw(IPC_PRIVATE IPC_RMID IPC_CREAT S_IRWXU);
use IPC::Semaphore;
sub MAXBUF() { 2000 }

sub alloc {             # метод конструктора
    my $class = shift();
    my $value = @_ ? shift() : "";

    my $key = shmget(IPC_PRIVATE, MAXBUF, S_IRWXU) || die "shmget: $!";
    my $sem = IPC::Semaphore->new(IPC_PRIVATE, 1, S_IRWXU | IPC_CREAT)
                        || die "IPC::Semaphore->new: $!";
    $sem->setval(0,1)   || die "sem setval: $!";

    my $self = bless {
        OWNER    => $$,
        SHMKEY   => $key,
        SEMA     => $sem,
    } => $class;

    $self->put($value);
    return $self;
}
```

Теперь о методах чтения и записи. Методы get и put блокируют буфер, а peek и poke – нет, поэтому последние два должны применяться, только когда объект заблокирован вручную, что приходится делать, если необходимо извлечь старое значение и записать обратно модифицированное – все в рамках одной и той же блокировки. Демонстрационная программа именно это и делает в цикле while (1). Транзакция целиком должна происходить под действием одной и той же блокировки, иначе проверка и установка не будут атомарными и способны стать бомбой.

```
sub get {
    my $self = shift();
    $self->lock;
    my $value = $self->peek(@_);
    $self->unlock;
    return $value;
}
sub peek {
    my $self = shift();
```

```
        shmread($self->{SHMKEY}, my $buff=q(), 0, MAXBUF) || die "shmread: $!";
        substr($buff, index($buff, "\0")) = q();
        return $buff;
    }
    sub put {
        my $self = shift();
        $self->lock;
        $self->poke(@_);
        $self->unlock;
    }
    sub poke {
        my($self,$msg) = @_;
        shmwrite($self->{SHMKEY}, $msg, 0, MAXBUF) || die "shmwrite: $!";
    }
    sub lock {
        my $self = shift();
        $self->{SEMA}->op(0,-1,0) || die "semop: $!";
    }
    sub unlock {
        my $self = shift();
        $self->{SEMA}->op(0,1,0) || die "semop: $!";
    }
```

Наконец, классу нужен деструктор, чтобы после того как объект закончит свое существование, можно было вручную освободить совместно используемую память и семафор, хранящийся внутри объекта. В противном случае они переживут своего создателя, и, чтобы избавиться от них, придется прибегнуть к *ipcs* и *ipcrm* (или помощи системного администратора). Вот почему в основной программе мы пошли на создание сложных оберток, преобразующих сигналы в исключительные ситуации: чтобы выполнились все деструкторы, из памяти были удалены объекты SysV IPC, а системные администраторы не занимались нашими делами.

```
    sub DESTROY {
        my $self = shift();
        return unless $self->{OWNER} == $$;  # избежать повтора освобождения памяти
        shmctl($self->{SHMKEY}, IPC_RMID, 0)   || warn "shmctl RMID: $!";
        $self->{SEMA}->remove()                || warn "sema->remove: $!";
    }
```

Сокеты

Механизмы IPC, обсуждавшиеся выше, имеют одно жесткое ограничение: они предназначены для организации связи между процессами, выполняющимися на одном и том же компьютере. (Несмотря на то что иногда доступ к файлам может осуществляться с нескольких машин, если используются механизмы типа NFS (сетевой файловой системы), во многих реализациях NFS блокировка не работает, что лишает одновременный доступ почти всей привлекательности.) Для организации сетевого взаимодействия общего назначения следует выбирать сокеты. Хотя сокеты (sockets) были придуманы для BSD, они быстро распространились на другие виды UNIX, а в наше время интерфейс сокетов можно найти почти в любой «жизнеспособной» операционной системе. Если на машине не поддерживаются сокеты, возникнут колоссальные трудности при работе с Интернетом.

С помощью сокетов можно создавать виртуальные контуры (в виде потоков TCP) и обмениваться датаграммами (в виде пакетов UDP). В зависимости от типа системы могут существовать и другие возможности. Но самый распространенный способ программирования сокетов – использование TCP через сокеты домена Интернета, о чем мы здесь и расскажем. Такие сокеты предоставляют надежные соединения, работающие подобно двунаправленным каналам, но не ограниченные локальной машиной. Оба главных приложения Интернета, электронная почта и браузеры, почти во всем полагаются только на сокеты TCP.

Вы также интенсивно используете UDP, не подозревая об этом. Всякий раз когда ваш компьютер пытается найти сайт в Интернете, он передает пакеты UDP серверу DNS, запрашивая фактический IP-адрес. Вы и сами можете использовать UDP, если захотите отправлять и принимать датаграммы. Датаграммы обходятся дешевле, чем соединения TCP, именно потому, что они не ориентированы на соединение; иначе говоря, они меньше похожи на звонок по телефону и больше – на попадание письма в почтовый ящик. Однако в UDP нет той надежности, которую предоставляет TCP, из-за чего UDP лучше подходит для ситуаций, когда неважно, что пара пакетов помялась, проколота или разорвана. Или когда известно, что некоторый протокол более высокого уровня обеспечит какую-то меру избыточности или амортизации отказов (именно так поступает DNS.)

Есть и другие возможности, но к ним обращаются значительно реже. Можно также использовать сокеты домена UNIX, но только для локального взаимодействия. Различные системы поддерживают другие протоколы, не основанные на IP. Несомненно, что где-то и для кого-то такие протоколы представляют интерес, но мы воздержимся от разговора о них.

Имена функций для сокетов в Perl совпадают с именами соответствующих системных вызовов C, но аргументы функций обычно отличаются. На то есть две причины: во-первых, дескрипторы файлов Perl действуют иначе, чем дескрипторы файлов в языке C; и, во-вторых, Perl уже знает длину своих строк, поэтому такую информацию передавать не нужно. Подробные сведения обо всех системных вызовах, касающихся сокетов, можно найти в главе 27.

Старый код на Perl с применением функций сокетов характеризует, в частности, проблема жестко закодированных значений констант, используемых при вызовах функций: такой подход уничтожает переносимость. Подобно большинству системных вызовов, функции сокетов тихо, но вежливо возвращают в случае неудачи `undef`, а не порождают исключение. Как следствие, крайне важно проверять возвращаемые значения этих функций, поскольку они не станут слишком шуметь, получив от вас мусор вместо нормальных аргументов. Встретив код, где, скажем, явным образом устанавливается значение `$AF_INET = 2`, имейте в виду – у вас крупные неприятности. Неизмеримо выше стоит подход, при котором используется модуль `Socket` или еще более дружелюбный модуль `IO::Socket` – оба стандартные. Эти модули предоставляют различные константы и вспомогательные функции для настройки клиентов и серверов. Наибольшего успеха программы, работающие с сокетами, добьются, если всегда будут начинаться следующим образом (и не забудьте ключ проверки меченых данных *-T* в строке `#!` для серверов):

```
#!/usr/bin/perl
use v5.14;
use warnings;
use autodie;
```

```
# или IO::Socket::IP из CPAN для поддержки IPv6
use IO::Socket;
```

Как отмечается в тексте книги, поведение Perl в плане взаимодействия с операционной системой в значительной мере зависит от библиотек C, и не все системы поддерживают все разновидности сокетов. Безопаснее всего придерживаться обычных операций через сокеты с TCP и UDP. Например, если нужна какая-то вероятность, что код удастся впоследствии перенести на системы, о которых никто не думал в момент разработки, не полагайтесь на поддержку надежного протокола упорядоченных пакетов. Также не следует передавать дескрипторы открытых файлов между неродственными процессами через локальный сокет домена UNIX. (Да, на многих UNIX-машинах это возможно – посмотрите страницу руководства *recvmsg*(2).)

Если требуется просто использовать стандартную службу Интернета, например электронную почту, телеконференции, службу доменных имен, FTP, Telnet, Web и т. д., не начинайте с чистого листа. Вместо этого попробуйте использовать модули CPAN. Для этих целей уже существуют такие модули, как Net::SMTP (или Mail::Mailer), Net::NNTP, Net::DNS, Net::FTP, Net::Telnet и различные модули поддержки HTTP. Комплекты модулей, libnet и libwww, содержат много отдельных модулей для работы в сети.

В последующих разделах мы представим несколько примеров клиентов и серверов, не слишком подробно объясняя каждую функцию, так как для этого пришлось бы повторить описания, имеющиеся в главе 27.

Сетевые клиенты

Используйте сокеты домена Интернета, если вам нужна надежная связь клиент-сервер между потенциально различными машинами.

Для создания клиента TCP, который соединяется с каким-то сервером, проще использовать стандартный модуль IO::Socket::INET:

```
#!/usr/bin/env perl
use v5.14;
use warnings;
use autodie;
use IO::Socket::INET;

my $remote_host = "localhost"; # замените действительным именем удаленного узла
my $remote_port = "daytime";   # замените названием службы или номером порта

my $socket = IO::Socket::INET->new(
                PeerAddr => $remote_host,
                PeerPort => $remote_port,
                Type     => SOCK_STREAM
            );

# послать что-то через сокет; в сети принято использовать CRLF
# служба daytime не принимает входные данные, но они могут требоваться другим службам
print $socket "Почему ты мне больше не звонишь?\r\n";

# прочесть ответ удаленной машины
my $answer = <$socket> =~ s/\R\z//r;
```

```
say "Получен ответ: $answer"

# и по окончании завершить соединение.
close($socket);
```

Если у нас есть имя узла и номер порта для подключения, а для остальных полей мы готовы оставить значения по умолчанию, достаточно сокращенной формы вызова:

```
$socket = IO::Socket::INET->new("www.yahoo.com:80")
    or die "Невозможно соединиться с портом 80 на yahoo: $!";
```

Для работы с протоколом IPv6 проще всего использовать модуль IO::Socket::IP из CPAN. В версиях Perl выше v5.14 этот модуль может быть даже включен в стандартную библиотеку. После того как вы получите указанный модуль, останется лишь изменить имя класса в примере выше с IO::Socket::INET на IO::Socket::IP, и он станет поддерживать также протокол IPv6. Этот класс обладает дополнительным методом sockdomain, который позволяет выяснить, какая используется версия протокола IP:

```
#!/usr/bin/env perl
use v5.14;
use warnings;
use autodie;
use IO::Socket::IP;

my $remote_host = "localhost";
my $remote_port = "daytime";

my $socket = IO::Socket::IP->new(
                PeerAddr => $remote_host,
                PeerPort => $remote_port,
                Type     => SOCK_STREAM,
             );

my $familyname = ( $socket->sockdomain == AF_INET6 ) ? "IPv6" :
                 ( $socket->sockdomain == AF_INET  ) ? "IPv4" :
                                                       "неизвестно";

say "Соединение с $remote_host:$remote_port установлено через $familyname";

# отправить что-нибудь через сокет: в сети принято использовать CRLF
print $socket "Почему ты мне не звонишь?\r\n";

# прочесть ответ удаленной машины
my $answer = <$socket> =~ s/\R\z//r;

say "Получен ответ: $answer";

# и по окончании завершить соединение.
close($socket);
```

Чтобы соединиться с помощью базового модуля Socket:

```
use v5.14;
use warnings;
use autodie;
```

```
use Socket;

my $remote_host = "localhost";
my $remote_port = 13;          # порт службы daytime

socket(my $socket, PF_INET, SOCK_STREAM, getprotobyname("tcp"));
my $internet_addr = inet_aton($remote_host);
my $paddr = sockaddr_in($remote_port, $internet_addr);

connect($socket, $paddr);
$socket->autoflush(1);

print $socket "Почему ты мне не звонишь?\r\n"
my $answer = <$socket> =~ s/\R\z//r;

say "Получен ответ: ". $answer;
```

Стандартный модуль Socket, входящий в состав v5.14, можно использовать и для работы с протоколом IPv6, но вызовы функций и API будут несколько отличаться от примера выше, который предназначен для работы исключительно с IPv4. Подробности читайте в разделе *Socket* справочного руководства.

Если необходимо закрыть соединение только со своей стороны, чтобы на удаленном конце был получен конец файла, но сохранить возможность чтения данных, поступающих с сервера, используйте системный вызов shutdown для полузакрытия:

```
# больше не выводить данные на сервер
shutdown($socket, 1); # константа Socket::SHUT_WR в v5.6
```

Сетевые серверы

В продолжение приведем соответствующий сервер. Его довольно просто реализовать посредством стандартного класса IO::Socket::INET:

```
use IO::Socket::INET;

$server = IO::Socket::INET->new(LocalPort => $server_port,
                                Type      => SOCK_STREAM,
                                Reuse     => 1,
                                Listen    => 10 ) # или SOMAXCONN
    || die "Невозможно стать сервером tcp на порту $server_port: $!\n";

while ($client = $server->accept()) {
    # $client является новым соединением
}

close($server);
```

Можно написать это и с помощью модуля более низкого уровня – Socket:

```
#!/usr/bin/env perl

use v5.14;
use warnings;
```

```
use autodie;
use Socket;

my $server_port = 12345; # выберите номер

# создать сокет
socket(my $server, PF_INET, SOCK_STREAM, getprotobyname("tcp"));

# чтобы можно было быстро перезапустить сервер
setsockopt($server, SOL_SOCKET, SO_REUSEADDR, 1);

# сконструировать адрес моего сокета
my $own_addr = sockaddr_in($server_port, INADDR_ANY);
bind($server, $own_addr);

# организовать очередь для входящих соединений
listen($server, SOMAXCONN);

# принимать и обрабатывать соединения
while (accept(my $client, $server)) {
    # сделать что-нибудь с новым соединением $client
} continue {
    close $client;
}

close($server);
```

Клиенту не требуется привязываться к какому-либо адресу вызовом `bind`, а вот серверу это нужно. Мы задали его адрес как `INADDR_ANY`, и это значит, что клиенты могут соединяться через любой имеющийся сетевой интерфейс. Чтобы работать через конкретный интерфейс (например, через внешний интерфейс машины-шлюза или брандмауэра), используйте действительный адрес этого интерфейса. (Клиенты тоже могут это делать, но им это редко требуется.)

Чтобы узнать, что за машина с вами соединилась, вызовите функцию `getpeername` для соединения с клиентом. Она возвращает IP-адрес, который придется самостоятельно перевести в имя (если получится):

```
use Socket;
$other_end = getpeername($client)
    || die "Невозможно идентифицировать клиента: $!\n";
($port, $iaddr) = unpack_sockaddr_in($other_end);
$actual_ip = inet_ntoa($iaddr);
$claimed_hostname = gethostbyaddr($iaddr, AF_INET);
```

Здесь имеется тривиальная возможность обмана, поскольку владелец этого IP-адреса может так настроить свои таблицы обратной трансляции адресов, что они будут говорить все что угодно. В качестве некоторой дополнительной меры надежности оттранслируйте результат в обратном направлении:

```
@name_lookup = gethostbyname($claimed_hostname)
    || die "Невозможно получить адрес $claimed_hostname: $!\n";
@resolved_ips = map { inet_ntoa($_) } @name_lookup[ 4 .. $#name_lookup ];
$might_spoof = !grep { $actual_ip eq $_ } @resolved_ips;
```

Когда клиент соединится с сервером, сервер сможет осуществлять ввод/вывод через дескриптор этого клиента. Но, пока сервер будет заниматься этим, он не сможет обслуживать последующие запросы, поступающие от других клиентов. Чтобы не ограничиваться обслуживанием единственного клиента, многие серверы немедленно создают свою копию вызовом `fork` **для обработки каждого входящего соединения. (Другие выполняют ветвление заранее или мультиплексируют ввод/вывод между несколькими клиентами посредством системного вызова** `select`**.)**

```
REQUEST:
while (accept(my $client => $server)) {
    if ($kidpid = fork) {
        close $client;     # родитель закрывает неиспользуемый дескриптор
        next REQUEST;
    }
    defined($kidpid) || die "невозможно выполнить fork: $!";

    close $server;         # потомок закрывает неиспользуемый дескриптор

    $client->autoflush(1);

    # ввод/вывод через указатель $client кода порожденного процесса
    # для каждого соединения
    $input = <$client>;
    print $client "output\n";  # или STDOUT, то же самое

    open(STDIN,  "<&", $client) || die "can't dup client: $!";
    open(STDOUT, ">&", $client) || die "can't dup client: $!";
    open(STDERR, ">&", $client) || die "can't dup client: $!";

    # запустим калькулятор, только для примера,
    system("bc -l");           # или что вам угодно, лишь бы не было
                               # управляющих символов для интерпретатора команд!
    print "done\n";            # все еще клиенту

    close $client;
    exit;                      # не пускать порожденный процесс назад к accept!
}
```

Для каждого поступившего запроса этот сервер запускает порожденный процесс через `fork`**. Благодаря этому он может одновременно обрабатывать много запросов, пока сохраняется возможность создавать новые процессы. (Возможно, вы захотите ограничить число одновременных запросов.) Даже если не выполнять** `fork`**, функция** `listen` **разрешает до** `SOMAXCONN` **(обычно пяти или более) ожидающих соединений. Каждое соединение использует некоторые ресурсы, хотя и в меньшем объеме, чем процесс. Серверы с разделением на процессы должны заботиться об уборке своих отработавших потомков (называемых «зомби» на жаргоне UNIX), иначе они быстро заполнят всю таблицу процессов. За вас это может сделать код** `REAPER`**, обсуждавшийся в разделе «Сигналы», либо вы сами можете присвоить** `$SIG{CHLD} = "IGNORE"`**.**

Прежде чем выполнить новую команду, мы подключаем стандартные устройства ввода, вывода (и вывода ошибок) к соединению с клиентом. В результате любая команда, читающая из `STDIN` **и пишущая в** `STDOUT`**, сможет общаться с удаленной**

машиной. Без переназначения команда не смогла бы найти дескриптор клиента, который по умолчанию закрывается при переходе через границу exec.

Если вы пишете сетевой сервер, мы настоятельно рекомендуем использовать ключ *-T* для активации проверки меченых данных, даже если вы не вызываете setuid или setgid. Это всегда полезно для серверов или любых других программ, выполняемых от чужого имени (как все сценарии CGI), поскольку уменьшает шансы посторонних проникнуть в вашу систему. Подробнее обо всем этом читайте в разделе «Обработка ненадежных данных» в главе 20.

Одно дополнительное соображение, касающееся создания программ для Интернета: многие протоколы требуют, чтобы концом строки служила комбинация CRLF, которую можно задать несколькими способами: "\r\n",[1] "\015\012", "\xd\xa" или даже chr(13).chr(10). На практике многие программы в Интернете применяют в качестве окончания строки просто "\012", но лишь потому, что программы для Интернета обычно стараются быть либеральными в отношении того, что они принимают, и строгими в отношении того, что выдают. (Если бы только люди поступали так же...)

Передача сообщений

Как упоминалось выше, связь по протоколу UDP требует значительно меньших накладных расходов, но не обеспечивает надежности, поскольку не гарантирует поступление сообщений в правильном порядке (и вообще их поступление). Часто говорят, что UDP означает Unreliable Datagram Protocol – ненадежный протокол датаграмм[2].

Тем не менее UDP имеет некоторые преимущества перед TCP, в том числе возможность широковещательной или групповой передачи сразу нескольким узлам (обычно в локальной сети). Если вы поймали себя на том, что излишне обеспокоены надежностью и начинаете встраивать проверки в свою систему передачи сообщений, то, возможно, следовало с самого начала использовать TCP. Правда, расходы на установление и разрыв соединения TCP будут больше, но если удастся компенсировать их передачей большого числа сообщений (или одного длинного сообщения), этим обстоятельством можно пренебречь.

Как бы то ни было, вот пример программы, использующей UDP. Она связывается с портом UDP службы времени на машинах, перечисленных в командной строке, или на всех, которые найдет с помощью широковещательного адреса, если аргументы не указаны.[3] Не на всех машинах включен сервер времени, особенно за границами брандмауэра, но в целом ожидается, что каждая вернет вам 4-байтное целое число (с порядком байтов в сетевом формате), представляющее текущее время по часам этой машины. Говоря точнее, возвращается число секунд, прошедших с 1900 года. Из этого числа нужно вычесть число секунд между 1900 и 1970 годами, чтобы передать это время функциям преобразования localtime или gmtime.

---

1 За исключением доисторических, появившихся еще до UNIX, систем Mac, которые, насколько нам известно, никто уже не использует.

2 В действительности: User Datagram Protocol – протокол пользовательских датаграмм. – *Прим. перев.*

3 Если это не работает, выполните команду *ifconfig -a*, чтобы определить правильный локальный широковещательный адрес.

```
#!/usr/bin/perl
# clockdrift - Сравнить показания часов в других системах с этой.
#             Если нет аргументов, передать время всем, кто слушает
#             Ждать ответа полсекунды.
use v5.14;
use warnings;
use strict;
use Socket;

unshift(@ARGV, inet_ntoa(INADDR_BROADCAST))
    unless @ARGV;

socket(my $msgsock, PF_INET, SOCK_DGRAM, getprotobyname("udp"))
    || die "сокет: $!";

# Требуется некоторым "неисправным" (borked)¹ машинам и не должно повредить остальным
setsockopt($msgsock, SOL_SOCKET, SO_BROADCAST, 1)
    || die "setsockopt: $!";

my $portno = getservbyname("time", "udp")
    || die "нет порта udp службы времени ";

for my $target (@ARGV) {
    print "Отправка $target:$portno\n";
    my $destpaddr = sockaddr_in($portno, inet_aton($target));
    send($msgsock, "x", 0, $destpaddr)
        || die "send: $!";
}

# служба времени возвращает 32-разрядное число секунд с начала 1900
my $FROM_1900_TO_EPOCH = 2_208_988_800;
my $time_fmt = "N";                       # причем в этом двоичном формате
my $time_len = length(pack($time_fmt, 1)); # годится любое число

my $inmask = q();              # строка для хранения битов fileno для select
vec($inmask, fileno($msgsock), 1) = 1;

# ждать появления входных данных полсекунды
while (select(my $outmask = $inmask, undef, undef, 0.5)) {
    defined(my $srcpaddr = recv($msgsock, my $bintime, $time_len, 0))
        || die "recv: $!";
    my($port, $ipaddr) = sockaddr_in($srcpaddr);
    my $sendhost = sprintf "%s [%s]",
                gethostbyaddr($ipaddr, AF_INET) || "UNKNOWN",
                inet_ntoa($ipaddr);
    my $delta = unpack($time_fmt, $bintime) -
                   $FROM_1900_TO_EPOCH - time();
    print "Часы $sendhost спешат на $delta секунд отн. данной системы.\n";
}
```

1 **Borked – эквивалент жаргонного слова «borken», представляющего собой намеренно искажаемое хакерами «broken». – *Прим. ред.***

16

Компиляция

Те, кто пришел сюда в поисках компилятора Perl, могут удивиться, обнаружив, что он у них уже есть. Программа *perl* (обычно */usr/bin/perl*) уже содержит компилятор Perl. Возможно, это не то, что вы думали, и если это так, вам будет приятно узнать, что мы предоставляем также *генераторы кода* (из лучших побуждений некоторые называют их «компиляторами»), которые обсудим ближе к концу главы. Но сначала поговорим о том, как мы представляем себе Компилятор. В этой главе нам неизбежно придется вникать в некоторые технические подробности, причем кому-то они будут интересны, а кому-то – нет. Если вы принадлежите к последним, то можете считать, что вам предоставлена возможность попрактиковаться в быстром чтении.

Представьте себя дирижером, заказавшим партитуру большого оркестрового произведения. Получив заказ, вы обнаруживаете несколько десятков книжечек, по одной на каждого участника оркестра, с партиями отдельных инструментов, а главный экземпляр, содержащий все партии, отсутствует. Хуже того, все партии, *которые есть*, написаны на обычном языке, а не с использованием нотной грамоты. Прежде чем составить программу выступления или даже раздать партии оркестрантам, необходимо перевести эти описания в прозе в обычную систему записи нотного стана. Затем составить из отдельных частей одну гигантскую партитуру, чтобы получить представление о произведении в целом.

Точно так же, если передать исходный код сценария Perl для выполнения программе *perl*, компьютер извлечет из него не больше пользы, чем музыканты из описания симфонии на обычном языке. Прежде чем программа начнет выполняться, Perl должен скомпилировать[1] эти похожие на английский язык указания в специальное символическое представление. Но и это еще не начало выполнения программы, потому что компилятор только компилирует. Подобно партитуре, лежащей перед дирижером, вашей программе, даже после преобразования в формат, пригодный для интерпретации, требуется активный агент, чтобы интерпретировать ее.

---

[1] Или транслировать, трансформировать, преобразовать, перевоплотить, превратить.

Жизненный цикл программ на Perl

Жизненный цикл программы на Perl можно разбить на четыре фазы, каждая из которых состоит из собственных этапов. Наибольший интерес представляют первая и последняя фазы, а две другие являются необязательными. Фазы изображены на рис. 16.1.

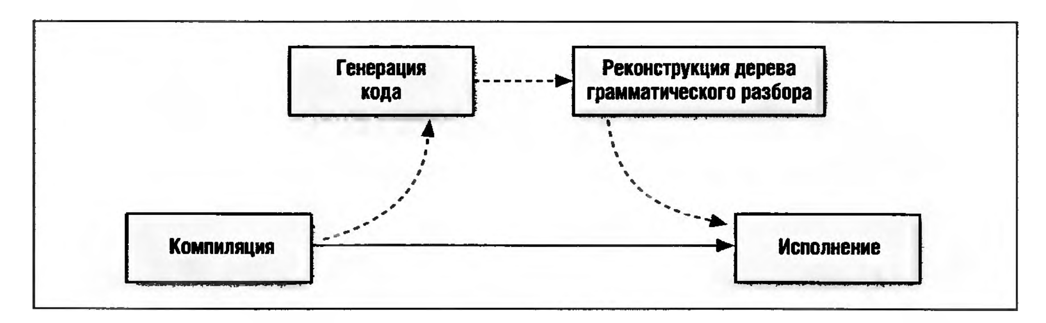


Рис. 16.1. *Жизненный цикл программы Perl*

1. *Фаза компиляции*

   Во время первой фазы, *фазы компиляции*, компилятор Perl преобразует программу в структуру данных, называемую *деревом грамматического разбора* (*parse tree*). Наряду со стандартной технологией грамматического разбора, Perl применяет другую, значительно более мощную: для управления дальнейшей компиляцией он использует блоки BEGIN. Они передаются интерпретатору сразу после разбора, и он выполняет их в порядке появления. В их число входят все объявления use и no, представляющие собой скрытые блоки BEGIN. Блоки UNITCHECK выполняются по окончании компиляции единицы компиляции – они используются для инициализации единиц компиляции. Выполнение всех блоков CHECK, INIT и END откладывается компилятором на более позднее время.

   Лексические объявления отмечаются, но присваивание в них не производится. На данном этапе компилируются все конструкции eval, *BLOCK*, s///e и неинтерполируемые регулярные выражения, и вычисляются константные выражения. Работа компилятора на этом заканчивается, если только не потребуется вызвать его позже. В конце этой фазы снова вызывается интерпретатор, чтобы выполнить запланированные блоки CHECK в порядке, обратном их поступлению. Наличие или отсутствие блока CHECK определяет, будет совершен переход к фазе 2 или скачок к фазе 4.

2. *Фаза генерации кода (необязательная)*

   Блоки CHECK устанавливаются генераторами кода, поэтому данная необязательная фаза имеет место в случае явного использования одного из генераторов кода, описываемых далее в разделе «Генераторы кода» этой главы. Они преобразуют скомпилированную, но еще не запущенную программу в исходный код на языке C или последовательность *байт-кодов* Perl – значений, выражающих внутренние инструкции Perl. Если вы решили создать исходный

код на C, то в конце концов из него можно создать файл на машинном языке,[1] называемый *исполняемым образом* (*executable image*). В этот момент ваша программа временно прекращает свои жизненные функции. После создания исполняемого образа можно сразу перейти к фазе 4; в противном случае потребуется подвергнуть полуфабрикат байт-кода восстановлению в фазе 3.

3. *Фаза реконструкции дерева грамматического разбора (необязательная)*

   Чтобы реанимировать программу, нужно воссоздать ее дерево грамматического разбора. Данная фаза наступает, только если производилась генерация кода и была выбрана генерация байт-кода. Perl должен реконструировать из этой последовательности свои деревья грамматического разбора, прежде чем сможет выполнить программу. Perl не выполняет байт-код непосредственно, это было бы слишком медленно.

4. *Фаза исполнения*

   Наконец то, чего все ждали: прогон программы. Поэтому данная фаза также носит название *фазы прогона* (*run phase*). Интерпретатор берет дерево грамматического разбора (полученное прямо от компилятора или косвенно через генерацию кода и последующую реконструкцию дерева грамматического разбора) и выполняет его. (Либо, если был создан исполняемый загрузочный модуль, он может выполняться как самостоятельная программа, поскольку в него встроен интерпретатор Perl.)

   В начале этой фазы, перед запуском основной программы, все запланированные блоки `INIT` выполняются в порядке следования в исходном коде. Затем выполняется основная программа. Интерпретатор может снова вызвать компилятор, встретив команду `eval` *`STRING`*, `do` *`FILE`*, оператор `require`, конструкцию `s///ee` или интерполируемое сопоставление с шаблоном, содержащее допустимое утверждение с кодом.

   По завершении основной программы будут выполнены отложенные блоки `END`, на этот раз в обратном порядке. Последним будет выполнен блок, встреченный первым, и на этом все закончится. Блоки `END` пропускаются только в случае вызова `exec` или если выполнение процесса будет прервано критической ошибкой. Обычные исключительные ситуации не считаются критическими.

Теперь обсудим эти фазы более подробно и в другом порядке.

Компиляция кода

Perl всегда находится в одном из двух режимов работы: либо компилирует программу, либо выполняет ее, и никогда в обоих состояниях одновременно. На протяжении этой книги мы часто говорим, что некоторые события происходят во время компиляции или что «*компилятор* Perl делает то-то и то-то». В других местах мы отмечаем, что нечто происходит во время выполнения или «*интерпретатор* Perl делает то-то и то-то». Хотя можно считать компилятор и интерпретатор просто частями «Perl»; осознание того, какую из двух ролей Perl играет в каждый

---

[1] Исходный сценарий тоже является *исполняемым файлом*, но не на машинном языке, поэтому мы не называем его образом. Файл образа носит такое название потому, что это дословная копия машинных кодов, которые выполняются непосредственно процессором.

конкретный момент, существенно для понимания причины возникновения тех или иных ситуаций. Исполняемый модуль *perl* играет обе роли: сначала компилятора, а затем интерпретатора. (Есть и другие роли: *perl* является также оптимизатором и генератором кода. Иногда даже проказником, но шутит только по-доброму.)

Важно понимать различие между фазой компиляции (compile phase) и временем компиляции (compile time), а также между фазой исполнения (run phase) и временем выполнения (run time). Типичная программа Perl проходит одну фазу компиляции, а затем одну фазу исполнения. «Фаза» – это широкое понятие, но время компиляции и время выполнения – понятия конкретные. В фазе компиляции производится в основном компилирование, но отчасти и выполнение – в блоках `BEGIN`. В фазе исполнения производится в основном выполнение, но могут осуществляться и действия времени компиляции, если встречаются операторы типа `eval` *`STRING`*.

В стандартной ситуации компилятор Perl сначала читает весь исходный текст программы, прежде чем начать выполнение. В это время производится синтаксический анализ объявлений, команд и выражений, проверяется их синтаксическая допустимость.[1] Найдя синтаксическую ошибку, компилятор пытается «не обращать на нее внимания», чтобы иметь возможность продолжить поиск ошибок в исходном тексте. Это удается не всегда: синтаксические ошибки имеют свойство порождать целый каскад ложных сообщений об ошибках. Встретив порядка 10 ошибок, Perl в отчаянии опускает руки.

В дополнение к интерпретатору, обрабатывающему блоки `BEGIN`, программу, при потворстве трех придирчивых агентов, обрабатывает компилятор. *Лексический анализатор* (*lexer*) высматривает в программе минимальные значимые единицы текста. В книгах, посвященных языкам программирования, их иногда называют *лексемами* (*lexemes*), или *маркерами* (*tokens*). Лексический анализатор иногда называется парсером, или сканером. Затем *парсер* (анализатор синтаксиса) пытается найти смысл групп этих лексем, собирая из них более крупные конструкции, такие как выражения и инструкции, допустимые в грамматике языка. *Оптимизатор* (*optimizer*) реорганизует эти более крупные образования и сокращает их в размерах, образуя более эффективные последовательности. Он тщательно выбирает, какую оптимизацию применить, не отвлекаясь на мелочи, поскольку компилятор Perl в режиме «загрузи и выполни» должен работать молниеносно.

Все это происходит не последовательно, а одновременно, при интенсивном общении агентов между собой. Лексическому анализатору иногда требуются советы синтаксического анализатора, чтобы определить, который из нескольких допустимых типов имеет рассматриваемая им лексема. (Как ни странно, лексическая область видимости (lexical scope) – одно из понятий, которые лексический анализатор *не понимает*, поскольку «лексический» имеет в данном случае другой смысл.) Оптимизатор тоже должен следить за действиями синтаксического анализатора, поскольку некоторые виды оптимизации нельзя применить, пока анализ не достигнет определенного момента, например не будет обнаружен конец выражения, инструкции, блока или подпрограммы.

---

[1] Формальных синтаксических диаграмм типа БНФ нет, но вы можете изучить файл *perly.y* в каталоге с исходными текстами Perl, содержащий грамматику *yacc*(1), используемую Perl. Советуем не трогать лексический анализатор, который известен тем, что вызывает нарушение аппетита у лабораторных крыс.

Может показаться странным, что компилятор Perl осуществляет все эти действия одновременно, а не одно за другим, но ведь это тот же запутанный процесс, который происходит при восприятии естественного языка на слух или во время чтения. Не нужно дожидаться конца главы, чтобы сообразить, каков смысл первого предложения. Такое соответствие представлено в табл. 16.1.

Таблица 16.1. Соответствие терминов в компьютерном и естественном языках

Компьютерный язык	Естественный язык
Символ	Буква
Лексема	Морфема
Терм	Слово
Выражение	Фраза
Инструкция	Предложение
Блок	Абзац
Файл	Глава
Программа	Повесть

Если синтаксический анализ проходит успешно, компилятор рассматривает исходный код как допустимую повесть – э-э, программу. Если программа запущена с ключом -*c*, выводится сообщение «syntax OK» и осуществляется выход. В противном случае компилятор передает плоды своих трудов другим агентам. Эти «плоды» имеют форму *дерева грамматического разбора* (*parse tree*). Каждый «плод» на этом дереве, или *узел* (*node*), как его называют, представляет один из *внутренних кодов операции* (*opcodes*) Perl, а ветви дерева представляют историческую структуру его роста. В конце концов, узлы будут связаны между собой линейно, один за другим, чтобы обозначить порядок, в котором система этапа исполнения должна посетить эти узлы.

Каждый код операции является наименьшей единицей представления выполняемых инструкций. Если вы считаете, что выражение `$a = -($b + $c)` – это одна инструкция, то Perl считает, что здесь шесть разных кодов операции. В упрощенном виде дерево грамматического разбора для этого выражения выглядит, как показано на рис. 16.2. Числа обозначают порядок, которому будет следовать система этапа выполнения.

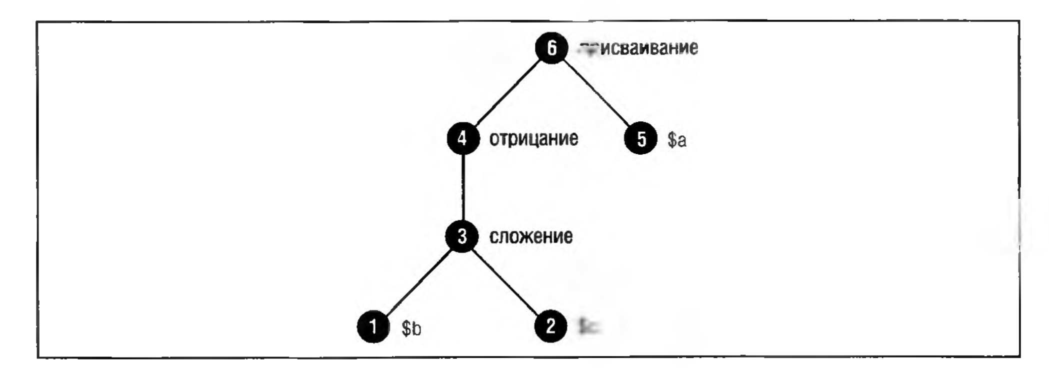


Рис. 16.2. Порядок выполнения кодов операций для выражения $a = –($b + $c)

Perl не является однопроходным компилятором, как могло показаться. (Однопроходные компиляторы сильно облегчают жизнь компьютеру, но осложняют ее программисту.) В действительности, это многопроходный оптимизирующий компилятор, осуществляющий по крайней мере три логически различных прохода, чередующихся между собой. Проходы 1 и 2 осуществляются поочередно по мере перемещения компилятора вверх-вниз по дереву синтаксического разбора во время его построения; проход 3 выполняется, когда подпрограмма или файл полностью разобраны. Вот описание этих проходов:

Проход 1: Восходящий синтаксический анализ

Во время этого прохода синтаксический анализатор *yacc*(1) строит дерево синтаксического разбора, используя лексемы, полученные от лексического анализатора (работу которого можно считать еще одним логическим проходом). «Восходящий» означает, что в поле зрения анализатора сначала попадают листья дерева, а уже потом ветви и корень. Он действительно производит вычисления снизу вверх, как видно на рис. 16.2, где мы изобразили корень вверху, в манере, характерной для ученых в области информатики (и лингвистов).

Для каждого создаваемого узла кода операции производится проверка правильности семантики, например, количество и типы аргументов при вызове встроенной функции. По мере формирования частей дерева оптимизатор рассматривает возможность преобразования расположенных ниже поддеревьев. Например, узнав, что функции передается конкретное число аргументов, он может отбросить код операции, запоминающей число аргументов, переданных функции с переменным числом аргументов. Более важный вид оптимизации, известный как *свертывание констант* (*constant folding*), описан далее в этом разделе.

На этом проходе определяется порядок посещения узлов при последующем выполнении. Это ловкий фокус, поскольку верхний узел почти никогда не является первым в списке посещений. Компилятор создает временный цикл из кодов операций, при этом верхний узел указывает на первый код операции, который необходимо посетить. Когда код операции верхнего уровня включается в более крупную структуру, цикл кодов операций разрывается и образуется больший цикл с новым узлом на вершине. В конечном итоге цикл разрывается совсем, когда начальный код операции вставляется в некую другую структуру, например описатель подпрограммы. При вызове подпрограммы все же можно найти тот первый код операции, несмотря на то, что он находится глубоко в структуре дерева, как на рис. 16.2. Интерпретатору нет необходимости возвращаться вниз по дереву синтаксического разбора, чтобы определить начальную точку.

Проход 2: Нисходящий оптимизатор

Читая отрывок кода на Perl (как и отрывок любого фрагмента текста), нельзя определить его контекст, не изучив окружающие лексические элементы. Иногда невозможно разобраться в происходящем без дополнительной информации. Не отчаивайтесь, однако, поскольку вы не одиноки: компилятору это тоже не удается. На этом проходе компилятор спускается вниз по только что построенному им поддереву, чтобы применить локальные оптимизации, самой примечательной из которых является *распространение контекста* (*context propagation*). Компилятор устанавливает для узлов, располагающихся ниже, надлежащий контекст (пустой, скаляр, список, ссылка или левостороннее значение), определяемый текущим узлом. Лишние коды операций обнуляются,

но не удаляются, поскольку менять порядок выполнения уже слишком поздно. Мы предоставим третьему проходу возможность удалить их из временного порядка выполнения, определенного на первом проходе.

Проход 3: Локальный оптимизатор

Некоторые участки кода имеют собственное пространство памяти, в котором хранятся переменные с лексическим контекстом. (На жаргоне Perl такая область называется *scratchpad*, или *временной памятью*.) В число этих участков входят инструкции `eval STRING`, подпрограммы и целые файлы. С позиций оптимизатора более важно, что все они имеют собственную точку входа. Хотя это и означает, что порядок выполнения от этой точки известен, но мы не можем знать, что происходило раньше, поскольку конструкция могла быть вызвана из любого места. Поэтому, произведя синтаксический анализ такого участка, Perl запускает для работы с ним локальный оптимизатор (peephole optimizer). В отличие от предыдущих двух проходов, когда осуществлялось перемещение по дереву синтаксического разбора, в этом проходе код рассматривается в линейном порядке, поскольку сейчас, в сущности, предоставляется последняя возможность сделать это перед тем, как оторвать список кодов операций от синтаксического анализатора. Большинство оптимизаций применяется на первых двух проходах, но некоторые оптимизации на этих проходах выполнить невозможно.

Здесь применяются различные завершающие оптимизации, в том числе устанавливается окончательный порядок выполнения путем пропуска обнуленных кодов операций, и определяются места, где те или иные сочетания кодов операций можно заменить чем-либо более простым. Важным видом оптимизации является распознавание следующих друг за другом операций конкатенации строк, чтобы избежать копирования строки всякий раз, когда к концу ее что-то добавляется. Этот проход не только оптимизирует, но и производит много «реальной» работы: отлавливает голые (bare) слова, генерирует предупреждения по поводу сомнительных конструкций, ищет код, который, вероятно, никогда не получит управления, выполняет подстановку ключей псевдохешей и отыскивает подпрограммы, вызванные до компиляции прототипов.

Проход 4: Генерация кода

Этот проход не является обязательным и обычно не используется. Но если вызван один из трех генераторов кода – `B::Bytecode`, `B::C` или `B::CC`, обход дерева синтаксического разбора выполняется еще раз. Генераторы кода создают либо последовательность байт-кодов Perl, используемую впоследствии для реконструкции дерева синтаксического разбора, либо код на C, отражающий состояние дерева синтаксического разбора на этапе компиляции.

Генерация кода на C может быть двух видов. `B::C` просто воссоздает дерево синтаксического разбора и запускает его с помощью обычного цикла `runops`, который сам Perl использует при выполнении. `B::CC` производит линеаризованный и оптимизированный C-эквивалент пути кода времени выполнения (напоминающий гигантскую таблицу переходов) и выполняет его.

Во время компиляции Perl применяет массу оптимизаций. Он перегруппировывает код, чтобы сделать его выполнение более эффективным. Удаляет код, который никогда не будет выполнен, например блоки `if (0)` или `elsif` и `else` в блоке `if (1)`. Если используются лексические типизированные переменные, объявленные как

my ClassName $var или our ClassName $var, и пакет ClassName был установлен с директивой use fields, обращения к значениям констант из псевдохешей во время компиляции проверяются на наличие опечаток и преобразуются в обращения к массивам. Если вы задаете в операторе sort достаточно простую процедуру сравнения, такую как {$a <=> $b} или {$b cmp $a}, она заменяется вызовом скомпилированного С-кода.

Наиболее впечатляющей из проводимых оптимизаций является ранняя подстановка константных выражений. Рассмотрим, например, дерево синтаксического разбора, изображенное на рис. 16.2. Будь узлы 1 и 2 литералами или постоянными функциями, узлы с 1 по 4 были бы заменены результатом вычисления этого выражения, как показано на рис. 16.3.



Рис. 16.3. *Свертывание констант*

Это называется *свертыванием констант* (*constant folding*). Свертывание констант не ограничивается такими простыми случаями, как, например, замена во время компиляции выражения 2**10 значением 1024. Оно включает также разрешение вызовов функций – как встроенных, так и объявляемых пользователями, – удовлетворяющих критериям, описанным в разделе «Подставляемые функции-константы» главы 7. Perl знает, какие из встроенных функций можно вызывать во время компиляции, что напоминает известную способность компиляторов FORTRAN вызывать свои внутренние функции. Поэтому если попытаться прологарифмировать 0.0 или извлечь корень из отрицательного числа, будет получена ошибка компиляции, а не времени выполнения, и интерпретатор вообще не будет запущен.[1] Даже выражения с произвольной степенью сложности вычисляются на ранней стадии, что иногда влечет удаление целых блоков, как, например, ниже:

```
if (2 * sin(1)/cos(1) < 3 && somefn()) { whatever() }
```

Perl не генерирует код для выражений, которые никогда не будут вычисляться. Поскольку левая часть условного выражения всегда принимает ложное значение, ни somefn, ни whatever никогда не будут вызваны. (Поэтому не рассчитывайте, что вам удастся перейти на метки, находящиеся внутри блока: во время выполнения они просто не будут существовать.) Будь somefn встраиваемой функцией-константой (inlinable constant), даже изменение порядка вычислений на такой:

```
if (somefn() && 2 * sin(1)/cos(1) < 3)) { whatever() }
```

не изменило бы результата, поскольку значение всего выражения по-прежнему можно определить во время компиляции. Будь whatever встраиваемой, ее вызов

---

[1] Мы упростили изложение. Интерпретатор запущен будет, поскольку это необходимо для свертывания констант. Но произойдет это непосредственно на этапе компиляции, подобно тому, как выполняются блоки BEGIN.

не производился бы ни во время выполнения, ни даже во время компиляции: ее значение было бы подставлено, как если бы функция являлась литеральной константой. Вы получили бы предупреждение о бесполезном применении константы в пустом контексте: «Useless use of a constant in void context». Оно может удивить, если не понять, что это константа. Однако будь whatever последней инструкцией в функции, вызываемой в непустом контексте (согласно определению оптимизатора), предупреждения вы бы не увидели.

Окончательный вид дерева синтаксического разбора после всех оптимизаций можно увидеть с помощью команды *perl -Dx*. (Ключ *-D* требует специальной версии Perl с поддержкой отладки). Читайте описание модуля B::Deparse в разделе «Средства разработки кода».

В общем, компилятор Perl старается (но *не слишком* сильно) оптимизировать код, чтобы, когда придет время запускать программу, она выполнялась быстрее. Пора бы уже и запустить вашу программу, чем мы сейчас и займемся.

Выполнение кода

В первом приближении Sparc-программы выполняются только на машинах Sparc, Intel-программы – только на машинах Intel, а Perl-программы – только на машинах Perl. Машина Perl обладает свойствами, которые Perl-программа считает идеальными для компьютера: автоматическое выделение и освобождение памяти, базовые типы данных в виде динамических строк, массивов и хешей без ограничений размера; а все системы ведут себя схожим образом. Задача интерпретатора Perl состоит в том, чтобы превратить компьютер, на котором он запущен, в одну из этих идеальных Perl-машин.

Эта фиктивная машина представляет собой иллюзию компьютера, специально разработанного исключительно для выполнения программ на Perl. Каждый код операции, порожденный компилятором, является базовой командой в этом эмулированном наборе инструкций. Вместо аппаратного счетчика команд за очередностью выполнения кодов операций следит интерпретатор. Вместо указателя стека интерпретатор имеет собственный виртуальный стек. Это очень важно, поскольку виртуальная машина Perl (мы отказываемся называть ее PVM)[1] является стековой. Коды операций Perl имеют внутреннее название *PP codes* («push-pop-коды»), поскольку для поиска операндов, обработки временных значений и хранения результатов они используют виртуальный стек интерпретатора.

Те, кому доводилось программировать на Forth или PostScript, либо работать с научным калькулятором HP с вводом в обратной польской (бесскобочной) нотации RPN (Reverse Polish Notation), знают, как работает машина со стековой организацией. Для тех, кто не знает, объясним. Идея проста: чтобы сложить 3 и 4, действия производятся в порядке 3 4 + вместо более привычного 3 + 4. На языке стека это означает, что сначала в стек проталкивается 3, затем 4, а потом + выталкивает со стека оба аргумента, складывает их и проталкивает 7 обратно в стек, где результат будет находиться, пока вы не сделаете с ним что-нибудь еще.

---

[1] По-видимому, чтобы никто не думал, что речь идет о параллельной виртуальной машине (тоже PVM) или о Pneumonia Virus of Mice. – *Прим. ред.*.

В сравнении с компилятором Perl интерпретатор Perl является простой и почти скучной программой. Он всего лишь поочередно перебирает скомпилированные коды операций и отправляет их среде выполнения Perl, то есть виртуальной машине Perl. Просто кусок кода на языке C, так?

На самом деле, этот кусок кода – довольно интересный. Виртуальная машина Perl поддерживает значительный объем динамического контекста, чтобы это не приходилось делать вам. Perl управляет множеством стеков, разбираться в которых не обязательно, но мы все же перечислим их здесь, чтобы произвести на читателя впечатление:

стек операндов

Об этом стеке мы уже говорили.

стек сохранения

Здесь сохраняются локализованные значения в ожидании восстановления. Многие внутренние процедуры локализуют значения незаметно для вас.

стек области видимости

Облегченный динамический контекст, определяющий, когда должны выталкиваться данные из стека сохранения.

стек контекста

Полновесный динамический контекст: последовательность вызовов, приведшая туда, где вы сейчас находитесь. Обход этого стека выполняет функция caller. Функции управления циклами просматривают этот стек в поисках цикла, которым нужно управлять. При перемещении назад по стеку контекста происходит соответствующее перемещение по стеку видимости, в результате чего восстанавливаются локальные переменные со стека сохранения, даже если выход из предыдущего контекста был выполнен каким-нибудь нехорошим способом вроде возбуждения исключительной ситуации и *longjmp*(3).

стек jumpenv

Стек контекстов *longjmp*(3), позволяющий возбуждать исключительные ситуации и выходить ускоренно.

стек возврата

Точка, из которой был выполнен вход в текущую подпрограмму.

стек маркеров (mark stack)

Место, где начинается текущий список аргументов на стеке операндов.

стеки временной памяти лексических переменных для рекурсии

Место хранения лексических переменных и других элементов «временного учета» при рекурсивном вызове подпрограмм.

И конечно, имеется стек C, где хранятся все переменные C. Perl старается не полагаться на стек C при хранении запоминаемых величин, поскольку при *longjmp*(3) не происходит правильного восстановления таких величин.

Все это сказано к тому, что обычная точка зрения на интерпретатор – программу, интерпретирующую другую программу, – оказывается, к сожалению, неадекватной для описания происходящего. Да, есть некоторый C-код, реализующий некоторые коды операций, но, произнося слово «интерпретатор», мы подразумеваем нечто большее – так же, как, произнося «музыкант», мы подразумеваем нечто

большее, чем набор инструкций ДНК для превращения нот в звуки. Музыканты – реальные живые организмы со своим «состоянием». То же относится к интерпретаторам.

Конкретно, весь этот динамический и лексический контекст, глобальные таблицы символов, деревья синтаксического разбора плюс поток выполнения и есть то, что мы называем интерпретатором. Как контекст выполнения, интерпретатор начинает существовать еще раньше, чем компилятор, и может существовать в рудиментарном виде даже в процессе создания компилятором контекста интерпретатора. На практике именно это и происходит, когда компилятор вызывает интерпретатор для выполнения блоков `BEGIN` и подобных им. А интерпретатор может воспользоваться компилятором, чтобы достраивать себя. Всякий раз, когда определятся новая подпрограмма или загружается новый модуль, виртуальная машина Perl, называемая интерпретатором, переопределяет себя заново. Невозможно точно сказать, которая из двух программ работает, компилятор или интерпретатор, потому что они взаимодействуют при управлении процессом самонастройки, который мы обычно называем «запуском сценария Perl». Это как самонастройка мозга ребенка. Кто ее осуществляет – ДНК или нейроны? Мы думаем – и ДНК, *и* нейроны, с некоторым участием привлеченных программистов.

В одном процессе можно запустить несколько интерпретаторов. При этом они могут совместно использовать деревья синтаксического разбора, если запускаются путем клонирования существующего интерпретатора. Можно также в одном интерпретаторе запускать несколько потоков, которые будут иметь общий доступ не только к деревьям синтаксического разбора, но и к глобальным символам.

Однако большинство программ на Perl использует только один интерпретатор Perl для выполнения скомпилированного кода. И хотя можно запустить несколько независимых интерпретаторов Perl в одном процессе, в настоящее время такая возможность доступна только из C. Каждый отдельный интерпретатор Perl играет роль отдельного процесса, но создание его обходится дешевле, чем для совершенно нового процесса. Вот почему модуль расширения Apache `mod_perl` достигает такой высокой производительности: при запуске сценария CGI под `mod_perl` этот сценарий оказывается уже скомпилированным в коды операций Perl, поэтому не требует перекомпиляции, но, что еще более важно, нет необходимости запускать новый процесс, а именно это и есть узкое место. Apache инициализирует новый интерпретатор Perl в существующем процессе и передает ему на выполнение ранее скомпилированный код. Конечно, как всегда, в действительности все сложнее.

Интерпретаторы Perl можно встраивать во многие приложения, например: *nvi*, *vim* и *innd*, – перечислить их все просто невозможно. Некоторые коммерческие продукты даже не упоминают о том, что в них встроено ядро Perl. Они просто используют Perl, поскольку он позволяет им красиво решать свои задачи.

Серверы компиляторов

Что ж, если Apache может скомпилировать программу на Perl и выполнить ее позже, почему вы не можете поступить так же? Apache и другие программы со встроенными интерпретаторами Perl живут просто: они не сохраняют дерево синтаксического разбора во внешнем файле. Если такой подход вас устраивает и вы

не возражаете против использования C API с этой целью, то можете осуществить то же самое.

Если вы не хотите идти таким путем или вам нужно что-то еще, есть несколько других вариантов. Вместо того чтобы передавать коды операций, созданные компилятором, непосредственно интерпретатору Perl, можно использовать другие серверы (backends). Они умеют сохранять скомпилированные коды операций во внешних файлах или даже преобразовывать их в C-код.

Имейте в виду, что эти генераторы кода являются в значительной степени экспериментальными, и в промышленной среде полагаться на них не следует. На самом деле, полагаться на них не следует и в непромышленной среде, и работать они будут от случая к случаю. Теперь, когда мы настроили вас так, что любой успех превзойдет всякие ваши ожидания. можно спокойно рассказать, как работают эти серверы.

Некоторые backend-модули являются генераторами кода, например `B::Bytecode`, `B::C` и `B::CC`. Другие, в сущности, являются средствами анализа и отладки, например `B::Deparse`, `B::Lint` и `B::Xref`. Помимо серверов (backends) в стандартную версию входит ряд других низкоуровневых модулей, возможно, представляющих интерес для будущих авторов средств разработки программ на Perl. Другие backend-модули можно найти в CPAN, в том числе (на момент написания данной книги) `B::Fathom`, `B::Graph` и `B::Size`.

Когда компилятор Perl используется не на входе интерпретатора, а в других целях, между ним и различными backend-модулями размещается модуль `O` (с помощью файла *O.pm*). Backend-модуль не вызывается напрямую, вместо него вызывается промежуточное звено, которое, в свою очередь, вызывает указанный сервер (backend). Если имеется, скажем, модуль `B::Backend`, то вызвать его с нужным сценарием можно так:

```
% perl -MO=Backend SCRIPTNAME
```

Некоторые backend-модули принимают параметры, которые указываются так:

```
% perl -MO=Backend, OPTS SCRIPTNAME
```

У некоторых серверов (backends) уже есть интерфейсы, позволяющие вызывать промежуточное звено, так что вам нет необходимости запоминать их М.О. В частности, *perlcc*(1) вызывает этот генератор кода, который может оказаться довольно неудобным для ручного запуска.

Генераторы кода

Со всей определенностью следует подчеркнуть, что все три сервера (backends) преобразования кодов операций Perl в некоторый другой формат являются экспериментальными. (Мы уже говорили об этом, но считаем необходимым напомнить еще раз.) Даже когда им случается сгенерировать код, который выполняется правильно, получившиеся программы могут потребовать больше дискового пространства, памяти и ресурсов CPU, чем в обычном случае. Исследования и разработка все еще продолжаются в этой области. Со временем положение будет улучшаться.

Генератор Bytecode

Модуль B::Bytecode выводит коды операций дерева синтаксического разбора в независимой от платформы кодировке. Сценарий на Perl, скомпилированный в байт-коды, можно перенести на любую машину, где установлен Perl.

Стандартная, но пока экспериментальная команда *perlcc*(1) позволяет преобразовать исходный код на Perl в скомпилированную в байт-коды программу Perl. Нужно лишь выполнить команду:

```
% perlcc -b -o pbyscript srcscript
```

Полученный сценарий *pbyscript* можно выполнить непосредственно. Начало этого файла выглядит примерно так:

```
#!/usr/bin/perl
use ByteLoader 0.03;
^C^@^E^A^C^@^@^@^A^F^@^C^@^@^@^B^F^@^C^@^@^@^C^F^@^C^@^@^@
B^@^@^@^H9^A8M-^?M-^?M-^?M-^?7M-^?M-^?M-^?M-^?6^@^@^@^A6^@
^G^D^D^@^@^@^KR^@^@^@^HS^@^@^@^HV^@M-2W<^FU^@^@^@^@X^Y@Z^@
...
```

Здесь виден небольшой заголовок сценария, за которым следуют двоичные данные. Это может выглядеть тайной магией, но в действительности, если магия здесь и есть, ее очень мало. Модуль ByteLoader использует технологию *входного фильтра* (*source filter*), чтобы преобразовать исходный текст, перед тем как передать его Perl. Входной фильтр – своего рода препроцессор, применяемый к содержимому текущего файла. Он выполняет не только простые преобразования, осуществляемые, например, макропроцессорами *cpp*(1) и *m4*(1): никаких ограничений не накладывается. Существуют входные фильтры для расширения синтаксиса Perl, сжатия и шифрования исходного кода и даже записи программ Perl на латыни. E perlibus unicode; cogito, ergo substr; carp dbm, et al. Э-э, caveat scriptor – пусть автор сценария будет бдителен.

Модуль ByteLoader – это входной фильтр, способный дизассемблировать переведенные в последовательную форму коды операций, произведенные B::Bytecode, и восстанавливать исходное дерево синтаксического разбора. Воссозданный код Perl образует текущее дерево синтаксического разбора без помощи компилятора. Когда интерпретатор находит эти коды операций, он просто выполняет их, как если бы они только того и ждали.

Генераторы кода на C

Оставшиеся генераторы кода, B::C и B::CC, создают не преобразованные в последовательную форму коды операций Perl, а код на языке C. Генерируемый ими код не отличается хорошей читаемостью, и вы ослепнете, если попытаетесь в нем разобраться. Использовать их, чтобы вставлять в большую программу на C маленькие фрагменты, оттранслированные из Perl в C, не удастся.

Модуль B::C просто выводит структуры данных на C, необходимые для воссоздания среды выполнения Perl целиком. В результате получается специализированный интерпретатор со всеми инициализированными структурами данных, построенными компилятором. В некоторых отношениях получаемый код похож на то, что создает B::Bytecode. Оба модуля выполняют простое транслирование деревьев кодов

операций, создаваемых компилятором, но если B::Bytecode делает это в символической форме, чтобы позже воссоздать и передать работающему интерпретатору Perl, то B::C записывает эти коды операций на C. Если скомпилировать этот код компилятором C и скомпоновать с библиотекой Perl, получившаяся программа будет выполняться без участия интерпретатора Perl. (Могут, однако, потребоваться некоторые общие библиотеки, если компоновка не была полностью статической.) Эта программа, в сущности, не будет отличаться от обычного интерпретатора Perl, выполняющего сценарий. Она просто будет скомпилирована как самостоятельный выполняемый модуль.

Модуль B::CC пытается пойти значительно дальше. Начало генерируемого им файла весьма похоже на то, что создает B::C,[1] но на этом всякое сходство заканчивается. В коде, создаваемом B::C, имеется большая таблица кодов операций на C, которая обрабатывается, как это делал бы сам интерпретатор, в то время как C-код, генерируемый B::CC, записывается в порядке, соответствующем логике выполнения вашей программы. В нем есть даже C-функции, соответствующие всем функциям вашей программы. Производится некоторая оптимизация по типам переменных. Некоторые контрольные тесты выполняются вдвое быстрее, чем при обычной интерпретации. Это наиболее претенциозный из имеющихся в настоящее время генераторов кода и самый многообещающий. Не случайно, что он наименее устойчив в работе.

Студенты, изучающие информатику и не выбравшие еще тему дипломного проекта, могут не искать ничего иного. Здесь масса необработанных алмазов, ждущих огранки.

Средства разработки кода

У модуля O есть много интересных возможностей помимо предоставления данных для невыносимо недоработанных генераторов кода. Предоставляя относительно легкий доступ к результатам работы компилятора Perl, этот модуль облегчает создание других инструментов, которым нужно знать все о программе на Perl.

Модуль B::Lint получил свое название от *lint*(1), верификатора программ C. Он ищет в программах сомнительные конструкции, о которые часто спотыкаются новички, но которые обычно не влекут вывод предупреждений. Модуль вызывается непосредственно:

```
% perl -MO=Lint,all myprog
```

В настоящее время осуществляется всего несколько видов проверок, например использование массивов в неявных скалярных контекстах, использование переменных по умолчанию и обращение к (номинально закрытым) начинающимся символом _ идентификаторам в других пакетах. Подробности читайте на страницах *B::Lint*(3). Вы наверняка встретитесь со многими программистами, проверяющими свои программы с помощью Perl::Critic. Это инструмент статического анализа, созданный на основе PPI и прекрасно справляющийся со своей работой.

[1] Как и все прочее, если вы потеряли зрение. Но мы же предупреждали, чтобы вы не подглядывали!

Модуль B::Xref генерирует списки перекрестных ссылок для объявлений и использования всех переменных (как с глобальной, так и с лексической областями видимости), подпрограмм и форматов с разбивкой по файлам и подпрограммам. Модуль вызывается так:

```
% perl -MO=Xref myprog > myprog.pxref
```

Вот пример фрагмента созданного отчета:

```
Subroutine parse_argv
  Package (lexical)
    $on              i113, 114
    $opt             i113, 114
    %getopt_cfg      i107, 113
    @cfg_args        i112, 114, 116, 116
  Package Getopt::Long
    $ignorecase      101
    &GetOptions      &124
  Package main
    $Options         123, 124  141, 150, 165, 169
    %$Options        141, 150, 165, 169
    &check_read      &167
    @ARGV            121, 157, 157, 162, 166, 166
```

Здесь видно, что в подпрограмме parse_argv – четыре собственных лексических переменных; кроме того, она обращается к глобальным идентификаторам из пакетов main и Getopt::Long. Числа – это номера строк, где используется объект: префикс i означает, что первое появление объекта имело место в строке с номером, который за ним следует, а префикс & указывает на вызов подпрограммы в соответствующей строке. Операции разыменовывания перечисляются отдельно и потому здесь показаны как $Options, так и %$Options.

Модуль B::Deparse позволяет снять пелену таинственности с кода Perl и понять, какие преобразования произвел над кодом оптимизатор. Ниже показано, например, какие значения по умолчанию использует Perl для различных конструкций:

```
% perl -MO=Deparse -ne 'for (1 .. 10) { print if -t }'
LINE: while (defined($_ = <ARGV>)) {
    foreach $_ (1 .. 10) {
        print $_ if -t STDIN;
    }
}
```

Ключ *-p* расставляет скобки, что позволяет видеть, какие приоритеты операторов использует Perl:

```
% perl -MO=Deparse,-p -e 'print $a ** 3 + sqrt(2) / 10 ** -2 ** $c'
print((($a ** 3) + (1.4142135623731 / (10 ** (-(2 ** $c))))));
```

С помощью ключа *-q* можно увидеть, в какие примитивы компилируются интерполируемые строки:

```
% perl -MO=Deparse,-q -e '"A $name and some @ARGV\n"'
'A ' . $name . ' and some ' . join($", @ARGV) . "\n";
```

А ниже показано, как Perl в действительности компилирует трехчастный цикл for в цикл while:

```
% perl -MO=Deparse -e 'for ($i=0;$i<10;$i++) { $x++ }'
$i = 0;
while ($i < 10) {
    ++$x;
}
continue {
    ++$i
}
```

Можно даже применить B::Deparse к файлу байт-кодов Perl, сгенерированному командой *perlcc -b*, и заставить его декомпилировать исполняемый модуль. Переведенные в последовательную форму коды операций Perl читать, может быть, трудновато, но вполне возможно.

Компилятор и интерпретатор: авангардизм и ретро

Думать обо всем следует в должное время: иногда заблаговременно, иногда – впоследствии. А бывает, что где-то между этими крайностями. Perl не притворяется, что знает, когда наступает подходящий момент для размышлений, поэтому предоставляет программисту ряд возможностей известить себя об этом. Иногда Perl понимает, что нужно немного подумать, но не знает, о чем именно, поэтому у него должен быть способ спросить об этом вашу программу, которая ответит на такие вопросы путем определения подпрограмм с именами, соответствующими тому, что Perl пытается выяснить.

Не только компилятор может вызвать интерпретатор, когда захочет быть предусмотрительным, но и интерпретатор может снова вызвать компилятор, когда ему нужно пересмотреть историю. Есть несколько операторов, которые можно использовать для обратного вызова компилятора. Как и компилятор, интерпретатор может вызывать подпрограммы по именам, когда ему нужно выяснить положение вещей. Весь этот обмен, происходящий между компилятором, интерпретатором и программой, требует, чтобы вы знали, когда и какие события происходят. Сначала мы обсудим вопрос о том, когда запускаются эти именованные подпрограммы.

В главе 10 мы видели, как запускается процедура AUTOLOAD при обращении к функции, которая не определена в этом пакете. В главе 12 мы познакомились с методом DESTROY, вызываемым, когда Perl намеревается автоматически освободить память, занимаемую объектом. И в главе 14 мы столкнулись со многими функциями, неявно вызываемыми при обращении к связанной переменной.

Все эти подпрограммы используют соглашение о том, что имена подпрограмм, автоматически запускаемых компилятором или интерпретатором, состоят из заглавных букв. С различными этапами жизненного цикла программы связаны еще пять таких подпрограмм: BEGIN, UNITCHECK, CHECK, INIT и END. Ключевое слово sub перед их объявлениями не обязательно. Возможно, правильнее было бы называть их «блоками», поскольку они ведут себя скорее как именованные блоки, а не как настоящие подпрограммы.

Например, в отличие от обычных подпрограмм, эти блоки допускается объявлять многократно, и вам не нужно вызывать их по имени, потому что Perl следит

за тем, когда их вызывать. (Отличие от обычных подпрограмм состоит также в том, что shift и pop действуют, как если бы вы находились в главной программе, то есть по умолчанию воздействуют на @ARGV, а не на @_.)

Эти пять блоков выполняются в следующем порядке:

BEGIN

Запускается ASAP (as soon as parsed – сразу после синтаксического разбора), как только встретится во время компиляции и перед компиляцией оставшейся части файла.

UNITCHECK

Запускается сразу после компиляции единицы компиляции, в которой определен. Главный файл программы и все модули, которые он загружает, являются единицами компиляции. Сюда же относятся строки eval, блоки кода в конструкциях (?{ }) и (??{ }) внутри регулярных выражений, вызовы do *FILE* и require *FILE*, и программный код, следующий за ключом *-e* в командной строке. UNITCHECK – это не INIT, который используется для определения программного кода, выполняющего инициализацию.

CHECK

Запускается по завершении компиляции, но перед запуском программы. (CHECK может означать «контрольную точку», «двойную проверку» или даже «останов».)

INIT

Запускается в начале выполнения перед началом основной логики программы.

END

Запускается в конце выполнения сразу после завершения программы.

Если имеется несколько таких подпрограмм с одинаковыми именами, даже в разных модулях, то все BEGIN выполняются перед всеми CHECK, которые все выполняются раньше всех INIT, которые выполняются раньше всех END, которые выполняются в самую последнюю очередь после завершения основной программы. Если блоков BEGIN и INIT несколько, они выполняются в порядке объявления (FIFO), а CHECK и END выполняются в порядке, обратном объявлению (LIFO).

Проще всего, вероятно, показать это на следующем примере:

```
use v5.10;
say       "начало выполнения main";
die       "завершение main\n";
die       "XXX: не достигнут\n";
UNITCHECK { say "1-й UNITCHECK: конец компиляции"    }
END       { say "1-й END: конец выполнения"          }
CHECK     { say "1-й CHECK: конец компиляции"        }
INIT      { say "1-й INIT: начало выполнения"        }
END       { say "2-й END: конец выполнения"          }
BEGIN     { say "1-й BEGIN: продолжение компиляции"  }
INIT      { say "2-й INIT: начало выполнения"        }
BEGIN     { say "2-й BEGIN: продолжение компиляции"  }
CHECK     { say "2-й CHECK: конец компиляции"        }
END       { say "3-й END: конец выполнения"          }
```

Эта демонстрационная программа выведет следующее:

```
1-й BEGIN: продолжение компиляции
2-й BEGIN: продолжение компиляции
1-й UNITCHECK: конец компиляции
2-й CHECK  конец компиляции
1-й CHECK: конец компиляции
1-й INIT: начало выполнения
2-й INIT: начало выполнения
начало выполнения main
завершение main
3-й END· конец выполнения
2-й END: конец выполнения
1-й END: конец выполнения
```

Поскольку блок BEGIN выполняется немедленно, он может объявлять, определять и импортировать подпрограммы еще до того, как будет скомпилирована остальная часть файла. В результате может измениться то, как компилятор осуществляет синтаксический анализ оставшейся части файла, особенно если импортируются определения подпрограмм. Как минимум, объявление подпрограммы позволяет использовать ее в качестве списочного оператора, делая необязательным применение скобок. Если импортированная подпрограмма объявлена с прототипом, обращения к ней могут интерпретироваться как вызовы встроенной подпрограммы и даже переопределять встроенные подпрограммы с тем же именем, сообщая им новую семантику. К объявлению use можно относиться просто как к блоку BEGIN.

Напротив, блоки END выполняются как можно *позже*: когда программа покидает интерпретатор Perl, даже в результате неперехваченного die или другой фатальной исключительной ситуации. Есть два случая, в которых блок END (или метод DESTROY) пропускается. Он не выполняется, если текущий процесс вместо выхода переходит из одной программы в другую посредством exec. Процесс, «выброшенный на берег» из-за неперехваченного сигнала, тоже пропускает свои процедуры END. (См. описание директивы sigtrap в главе 29, где рассматривается преобразование перехваченных сигналов в исключительные ситуации. Общая информация по обработке сигналов дана в разделе «Сигналы» главы 15.) Предотвратить выполнение всех END можно, вызвав POSIX::_exit, выполнив команду *kill -9 $$*[1] или просто запустив через exec какую-нибудь безобидную программу вроде */bin/true* в системах UNIX.

Внутри блока END переменная $? содержит код состояния, с которым программа собирается выйти. Значение $? можно изменить внутри блока END, чтобы программа завершилась с другим кодом состояния. Не измените случайно $?, запустив другую программу через system или обратные апострофы.

Если в файле имеется несколько блоков END, они выполняются в порядке, обратном порядку их определений. То есть блок END, объявленный последним, будет выполнен первым по завершении программы. Это позволяет организовать правильное вложение попарно связанных друг с другом блоков BEGIN и END. Например, если основная программа и загружаемый ею модуль имеют свои парные подпрограммы BEGIN и END, как показано ниже:

---

[1] Имеется в виду, вызвав функцию system("kill -9 $$") из программы. – *Прим. перев.*

```
BEGIN { print "main begun" }
END   { print "main ended" }
use Module;
```

и такие объявления в модуле:

```
BEGIN { print "module begun" }
END   { print "module ended" }
```

то основная программа знает, что ее BEGIN всегда выполнится первым, а ее END – последним. (Да, BEGIN является, в сущности, блоком этапа компиляции, но аналогичные принципы применимы к парным блокам INIT и END на этапе выполнения.) Этот принцип рекурсивно выполняется для любого файла, содержащего другой, если в обоих есть такие объявления. Данное свойство вложенности позволяет использовать блоки как конструкторы и деструкторы пакетов. В каждом модуле могут быть свои функции установки и демонтажа, которые Perl вызовет автоматически. Благодаря этому программисту, применяя конкретные библиотеки, не нужно помнить, какой код инициализации или уборки мусора должен вызываться и когда. Объявления в модуле обеспечивают эти события.

Если рассматривать eval *STRING* как *обратный вызов* интерпретатором компилятора, то BEGIN можно рассматривать как *прямой вызов* компилятором интерпретатора. В обоих случаях текущая деятельность приостанавливается, и переключается режим работы. Когда мы говорим, что блок BEGIN выполняется как можно раньше, мы имеем в виду, что он выполняется, как только будет полностью определен, даже раньше, чем будет проанализирована оставшаяся часть файла. Поэтому блоки BEGIN всегда выполняются на этапе компиляции, а не выполнения. После выполнения блока BEGIN он сразу становится неопределенным, а использовавшийся в нем код возвращается в пул памяти Perl. При всем желании вызвать блок BEGIN как подпрограмму не удастся, потому что к тому времени, когда вы сможете это сделать, его уже не будет в живых.

Аналогично блокам BEGIN, блоки INIT запускаются перед началом этапа выполнения Perl в порядке «первым пришел, первым ушел» (FIFO). Например, генераторы кода, описанные в странице руководства *perlcc*, используют блоки INIT для инициализации и разрешения указателей на XSUB. Блоки INIT очень похожи на блоки BEGIN, но позволяют программисту отделить конструктор, выполняемый на этапе компиляции, от конструктора, выполняемого на этапе выполнения. При непосредственном запуске сценария это не очень важно, поскольку в любом случае каждый раз вызывается компилятор. Но если компиляция отделена от выполнения, это различие может быть критическим. Компилятор может быть вызван один раз, а полученный выполняемый модуль может вызываться многократно.

Аналогично блокам END, блоки CHECK запускаются сразу по окончании этапа компиляции Perl, но до начала этапа выполнения, в порядке, обратном их следованию (LIFO). Блоки CHECK могут пригодиться, чтобы «свернуть» компилятор, подобно тому, как блоки END удобно применять, чтобы «свернуть» программу. В частности, серверы используют блоки CHECK как перехватчики для запуска своих генераторов кода. Все, что им нужно сделать – это вставить блок CHECK в собственный модуль, и он будет запущен в нужный момент, а вам не потребуется вставлять CHECK в свою программу. Поэтому программисту редко требуется самому писать блок CHECK, если только он не создает свой модуль такого типа.

Объединяя все сказанное, табл. 16.2 перечисляет различные конструкции с подробностями о том, когда они компилируются и когда выполняют код, обозначенный «...».

Таблица 16.2. Что когда происходит[1]

Блок или выражение	Компилируется в фазе	Перехватывает ошибки компиляции	Выполняется в фазе	Перехватывает ошибки выполнения	Политика запуска вызова
`use ...`	C	Нет	C	Нет	Сейчас
`no`	C	Нет	C	Нет	Сейчас
`BEGIN {...}`	C	Нет	C	Нет	Сейчас
`UNITCHECK {...}`	C	Нет	C	Нет	Поздний
`CHECK {...}`	C	Нет	C	Нет	Поздний
`INIT {...}`	C	Нет	R	Нет	Ранний
`END {...}`	C	Нет	R	Нет	Поздний
`eval {...}`	C	Нет	R	Да	Встроенный
`eval "..."`	R	Да	R	Да	Встроенный
`foo(...)`	C	Нет	R	Нет	Встроенный
`sub foo {...}`	C	Нет	R	Нет	Вызов в любое время
`eval "sub {...}"`	R	Да	R	Нет	Поздний вызов
`s/pat/.../e`	C	Нет	R	Нет	Встроенный
`s/pat/"..."/ee`	R	Да	R	Да	Встроенный

Теперь, уяснив партитуру, вы сможете сочинять и исполнять свои пьесы для Perl с большей уверенностью.

[1] Символом «C» в таблице обозначается фаза компиляции (Compile), а символом R – фаза выполнения (Runtime). – *Прим. перев.*

17

Интерфейс командной строки

Эта глава посвящена тому, как наводить орудие Perl перед выстрелом. Есть несколько способов наводки Perl. Два основных – ключи командной строки и переменные среды. Ключи обеспечивают более сподручный и точный способ нацеливания конкретной команды. Переменные среды чаще используются для настройки общей политики.

Обработка команд

Большая удача, что Perl эволюционировал в мире UNIX, поскольку, благодаря этому, синтаксис его вызова работает достаточно хорошо в интерпретаторах команд и других операционных систем. Большинство интерпретаторов команд в состоянии интерпретировать список слов как аргументы и не смущается, если первый символ аргумента начинается символом «–». Есть, правда, неприятные моменты, в которых можно запутаться при переходе от одной системы к другой. Например, в MS-DOS нельзя использовать одиночные кавычки, как это делается в UNIX. А в таких системах, как VMS, некоторым программам-оберткам придется нелегко при эмулировании переадресации ввода/вывода UNIX. Интерпретация масок – вообще дело непредсказуемое. Однако после того, как вы разберетесь с этими тонкостями, Perl станет работать с ключами и аргументами сходным образом в любой операционной системе.

Даже если у вас отсутствует собственно интерпретатор команд, программы на Perl легко можно запускать из любых других программ, написанных на любом языке. Вызывающая программа может передавать аргументы не только обычным способом, но и через переменные среды и, если позволяет операционная система, через унаследованные дескрипторы файлов (см. раздел «Передача дескрипторов файлов» главы 15). Даже самые экзотические механизмы передачи аргументов легко можно реализовать в отдельном модуле, подключив его к программе Perl с помощью обычной директивы use.

Perl анализирует аргументы командной строки стандартным образом. Это значит, что он рассчитывает получить ключи (слова, начинающиеся со знака «--»)

в начале командной строки. Затем обычно следует название файла сценария, а за ним – дополнительные аргументы, которые ему нужно передать. Некоторые из этих дополнительных аргументов сами могут выглядеть как ключи, но их должен обрабатывать сценарий, поскольку Perl прекращает разбор ключей, встретив аргумент, не являющийся ключом, или специальный ключ «--», означающий, что этот ключ – последний.

Perl дает некоторую свободу в размещении исходного кода программы. Если речь о маленькой одноразовой задаче, программу на Perl можно целиком уместить в командной строке. Более объемные и «долгоживущие» задачи можно оформлять как сценарии Perl в отдельных файлах. Сценарий для компиляции и запуска Perl может быть указан одним из следующих трех способов:

1. Построчно в командной строке с помощью ключей *-e* или *-E*, например:

   ```
   % perl -e "print 'Hello, World.'"
   Hello, World.

   % perl -E "say 'Howdy y\\'all!'"
   Howdy y'all!
   ```

2. По имени файла, содержащего сценарий: в качестве такового используется первый файл, обозначенный в командной строке. Системы, поддерживающие обозначение #! в первой строке файла с выполняемым сценарием, вызывают интерпретатор, указанный за данным обозначением.

3. Неявно, через стандартный ввод. Этот метод срабатывает только при отсутствии имен файлов среди аргументов; чтобы передать аргументы сценарию, принятому из стандартного ввода, нужно использовать метод 2, явно указав в качестве имени сценария «-». Например:

   ```
   % echo "print qq(Hello, @ARGV.)" | perl - World
   Hello, World.
   ```

При использовании методов 2 и 3 Perl начинает синтаксический анализ входного файла с самого начала, но если задан ключ *-x*, тогда он ищет первую строку, начинающуюся с #! и содержащую слово «perl», и начинает анализ с этого места. Это полезно при запуске сценария, встроенного в текст длинного сообщения. Обозначить конец сценария в этом случае можно с помощью маркера `__END__`.

Используете вы ключ *-x* или нет, но при синтаксическом анализе в строке #! всегда выполняется поиск ключей. Благодаря этому при работе на платформе, поддерживающей только один аргумент в строке #! или, того хуже, даже не воспринимающей строку #! как особую, можно иметь единообразное восприятие ключей независимо от того, как был вызван Perl, даже если для поиска начала сценария был указан ключ *-x*.

Предупреждение: поскольку в ранних версиях UNIX ядро молча заканчивает интерпретацию строки #! после 32 символов, некоторые ключи программа может получить в целости, а другие – нет; можно даже получить «-» без последующей буквы, если не проявить осторожность. Возможно, потребуется сделать так, чтобы все ключи находились по одну сторону (до или после) этой границы в 32 символа. Для большинства ключей неважно, если они обработаются лишний раз, но получение «-» вместо целого ключа приведет к тому, что Perl будет пытаться прочесть исходный код со стандартного ввода, а не из вашего сценария. Неполный ключ *-I* тоже может привести к неожиданным результатам. Однако некоторые ключи не

безразличны к повторной обработке, например комбинации *-l* и *-0*. Либо поместите все ключи за 32-символьной границей (если это возможно), либо вместо *-0DIGITS* используйте `BEGIN{ $/ = "\0DIGITS"; }`. Конечно, если вы работаете не под UNIX, то с такой проблемой вы не столкнетесь.

Синтаксический разбор ключей в строке `#!` начинается с места, где впервые встретится слово «perl». Последовательности «-*» и «- » специально игнорируются в интересах пользователей *emacs*, поэтому при желании можно записать:

```
#!/bin/sh -- # -*- perl -*- -p
eval 'exec perl -S $0 ${1+"$@"}'
    if 0;
```

и Perl распознает только ключ `-p`. Причудливое сочетание «`-*- perl -*-`» сообщает *emacs* о необходимости начать работу в режиме Perl; если вы не пользуетесь *emacs*, оно вам не потребуется. Путаница с ключом *-S* разъясняется ниже, в описании этого ключа.

Аналогичный трюк связан с программой *env*(1), если она у вас есть:

```
#!/usr/bin/env perl
```

В предыдущих примерах используется относительный путь к интерпретатору Perl, находящий ту его версию, которая первой встретится в маршруте поиска пользователя. Если требуется особая версия Perl, например *perl5.14.0*, поместите ее прямо в путь строки `#!`, независимо от того, используете ли вы программу *env*, ключ *-S* или обычную обработку `#!`.[1]

Если строка `#!` *не* содержит слова «perl», вместо интерпретатора Perl будет выполнена программа, указанная после `#!`. Предположим, что у вас есть обычный сценарий интерпретатора команд Борна с таким текстом:

```
#!/bin/sh
echo "I am a shell script"
```

Если подать этот файл на вход Perl, он запустит */bin/sh*. Это несколько странно, но полезно на машинах, которые не распознают `#!`, поскольку путем установки переменной среды `SHELL` можно сообщить программе (например, почтовой), что интерпретатором команд служит */usr/bin/perl*, и Perl отправит программу нужному интерпретатору, даже если ядро не сможет этого сделать само.

Но вернемся к сценариям Perl, действительно являющимся сценариями Perl. Отыскав сценарий, Perl скомпилирует всю программу во внутреннее представление (см. главу 16). Если возникнут ошибки компиляции, выполнение не начнется. (В то время как обычные сценарии интерпретатора команд и командные файлы могут частично выполниться, прежде чем будет обнаружена синтаксическая ошибка.) Если сценарий синтаксически корректен, он выполняется. Если сценарий выполнен до конца, и при этом не встретились операторы `exit` или `die`, Perl неявно вызывает `exit(0)`, сообщая вызывающей программе об успешном выполнении. (Такое поведение несвойственно обычным программам на языке C, которые могут вернуть случайное число при завершении без кода возврата.)

---

[1] Возможно, будет удобнее использовать для переключения версий специально предназначенную для этого программу perlbrew. – *Прим. науч. ред*

#! и кавычки в системах, отличных от UNIX

Механизм #!, используемый в UNIX, можно эмулировать в других системах:

MS-DOS

Создайте пакетный файл для запуска вашей программы и закодируйте его согласно ALTERNATIVE_SHEBANG. Дополнительные сведения можно почерпнуть из файла *dosish.h* в корневом каталоге дистрибутива с исходным кодом Perl.

OS/2

Поместите строку:

```
extproc perl -S -your_switches
```

первой в файле **.cmd* (*-S* позволяет обойти ошибку обработки «extproc» в *cmd.exe*).

VMS

Поместите в начале программы строки:

```
% perl -mysw 'f$env("procedure")' 'p1' 'p2' 'p3' 'p4' 'p5' 'p6' 'p7' 'p8' !
$exit++ + ++$status != 0 and $exit = $status = undef;
```

где *-mysw* обозначает ключи командной строки, которые нужно передать Perl. После этого можно вызвать программу как процедуру DCL (команда @program), непосредственно введя с клавиатуры perl program, или неявным образом через DCL$PATH, используя лишь имя программы. Это «заклинание» немного сложно запомнить, но Perl отобразит его, если вы введете perl "-V:startperl". Если вам и этого не запомнить... что ж, затем вы и купили эту книгу.

Windows

При использовании дистрибутива ActiveState Perl в какой-либо версии операционной системы Microsoft Windows (а именно Win95, Win98, Win00,[1] WinNT, но не Win3.1), процедура установки Perl модифицирует реестр Windows, связывая расширение *.pl* с интерпретатором Perl.

При установке версии Perl от другого производителя, включая версии для Win32 с официального сайта, модифицировать реестр Windows придется вручную.

Обратите внимание, что при использовании расширения *.pl* Windows не сможет отличить выполняемую программу Perl от файла «библиотеки perl». Чтобы избежать этого, можно использовать для программ Perl расширение *.plx*. В последнее время масштабы этого бедствия уменьшились, поскольку большинство модулей Perl теперь имеет расширение *.pm*.

Интерпретаторы команд в системах, отличных от UNIX, часто имеют совершенно другие представления о применении кавычек. Необходимо знать, какие специальные символы использует интерпретатор команд (обычно это *, \ и ") и как представлять пробельные символы и специальные символы, чтобы запускать однострочные программы посредством ключа *-e*. Может также потребоваться заменить одиночный % на %% или преобразовать его иным образом, если символ процента является для вашего интерпретатора команд специальным.

---

[1] Приносим извинения за технические неполадки...

В некоторых системах может потребоваться заменить одиночные кавычки двойными. Только не делайте этого в системах UNIX или Plan9, а также в интерпретаторах команд в стиле UNIX, таких как системы из MKS Toolkit или пакета Cygwin, производимого ребятами из Cygnus, входящей теперь в Redhat. Новый эмулятор UNIX под названием Interix, разработанный в Microsoft, тоже начинает выглядеть э-э... интериксно.

Например, в UNIX (включая Linux и OS X) командная строка будет выглядеть так:

```
% perl -e 'print "Hello world\n"'
```

Вид командной строки в VMS:

```
$ perl -e "print ""Hello world\n"""
```

или с qq//:

```
$ perl -e "print qq(Hello world\n)"
```

А в MS-DOS и прочих:

```
A:> perl -e "print \"Hello world\n\""
```

или с qq// для выбора собственных кавычек:

```
A:> perl -e "print qq(Hello world\n)"
```

Проблема в том, что ни один из этих способов не надежен: его работоспособность зависит от используемого интерпретатора команд. Здесь нет общего решения, и царит полнейшая неразбериха. Если вы работаете в системе, отличной от UNIX, но вам нравится командная строка, постарайтесь приобрести интерпретатор команд получше того, что установлен изначально, это не должно оказаться сложным делом.

Или пишите все на Perl и забудьте об однострочниках.

Местонахождение Perl

Это может казаться очевидным, но Perl полезен, только когда пользователь легко может найти его. Если это возможно, хорошо, чтобы */usr/bin/perl* и */usr/local/bin/perl* были символическими ссылками на фактический исполняемый модуль. Если это невозможно, системному администратору настоятельно рекомендуется поместить Perl и сопутствующие утилиты в каталог, входящий в стандартный путь поиска PATH пользователя или другое удобное место.

В данной книге мы используем стандартное обозначение #!/usr/bin/perl в первой строке программы – независимо от конкретного механизма, действующего в системе. Чтобы запустить конкретную версию Perl, используйте точный путь:

```
#!/usr/local/bin/perl5.14.0
```

Если вы хотите запустить версию *не ниже* некоторого номера, но не возражаете против более высоких номеров, поместите такую команду в начале программы:

```
use v5.14.0;
```

(Замечание: ранние версии Perl используют такие номера, как 5.005 или 5.004_05. Сегодня мы считали бы их версиями v5.5.0 и v5.4.5, но версии Perl старше v5.6.0 не понимают таких обозначений. Форма use 5.*NNN* является самой безопасной и гарантирует обратную совместимость с продуктами из глубин прошлого тысячелетия.)

Ключи

Ключ командной строки из одного символа без аргумента всегда можно связать с ключом, который следует за ним.

```
#!/usr/bin/perl -spi.bak # то же, что и -s -p -i.bak
```

Ключи (*switches*) называют также *опциями* (*options*), или *флагами* (*flags*). Как бы вы их ни называли, вот как их понимает Perl:

`--`
: Прекращает обработку ключей, даже если следующий аргумент начинается знаком «минус». Другого действия не оказывает.

-0DIGITS

Задает разделитель входных записей ($/) в виде восьмеричного или шестнадцатеричного числа, представляющего код символа. Если *DIGITS* отсутствует, разделителем становится символ NUL (т.е. U+0000, "\0" в Perl). До или после числа могут находиться другие ключи. Например, если в системе установлена версия *find*(1), способная выводить имена файлов, оканчивающиеся нулевым символом, можно написать так:

```
% find . -name '*.bak' -print0 | perl -n0e unlink
```

Особое значение 00 заставляет Perl читать файлы в режиме абзацев, что эквивалентно установке переменной $/, равной "". А любое значение от 0400 до 0777 заставляет проглатывать сразу целые файлы, но в соответствии с соглашениями обычно используется значение 0777, что эквивалентно неопределенному значению переменной $/. Мы используем 0777, поскольку символа ASCII с таким значением нет. (К несчастью, символ Юникода с таким значением *существует*: LATIN SMALL LETTER O WITH STROKE AND ACUTE – маленькое латинское «о» со штрихом и акутом, но что-то подсказывает нам, что вы не станете использовать его в качестве разделителя своих записей. Если же внезапно вам это потребуется, просто укажите его код в шестнадцатеричном виде: *-0x1FF*.)

Символ-разделитель можно также указать в шестнадцатеричном виде: *-0xHHH...*, где каждый символ «H» обозначает допустимую шестнадцатеричную цифру. В отличие от восьмеричной формы записи, данная форма может использоваться для определения любых символов Юникода, даже находящихся за границей 0xFF. (Это означает, что вы не сможете использовать ключ *-x* с именем каталога, состоящим только из шестнадцатеричных цифр.)

-a
: Включает режим авторазбивки, но только когда используется в сочетании с ключом *-n* или *-p*. В неявном цикле while, организуемом ключами *-n* и *-p*, сначала неявно вызывается функция split, разбивающая строку в массив слов @F. Поэтому команда

```
% perl -ane 'print pop(@F), "\n";'
```

эквивалентна:

```
LINE: while (<>) {
    @F = split(' ');
    print pop(@F), "\n";
}
```

Задать другой разделитель полей можно с помощью ключа *-F*, указав в нем регулярное выражение для split. Например, следующие вызова эквивалентны:

```
% awk -F: '$7 && $7 !~ /^\/bin/' /etc/passwd
% perl -F: -lane 'print if $F[6] && $F[6] !~ m(^/bin)' /etc/passwd
```

-c Заставляет Perl проверить синтаксис сценария, а затем завершить работу, не выполняя скомпилированный код. Технически этот ключ не ограничивается проверкой синтаксиса: выполняются все блоки BEGIN, UNITCHECK и CHECK, а также директивы use и no, поскольку считается, что они обрабатываются до выполнения программы. Однако блоки INIT и END не выполняются. Прежний режим (который редко оказывается полезен) можно получить, поместив в конце основного сценария:

```
BEGIN { $^C = 0; exit; }
```

Дело здесь в том, что переменная $^C отражает значение ключа *-c*.

-C[number|list]

Ключ *-C* управляет некоторыми особенностями Perl, связанными с обработкой Юникода. За ключом *-C* может следовать число или список букв. Допустимые буквы, их числовые значения и оказываемые эффекты перечислены в табл. 17.1; список букв эквивалентен сумме соответствующих чисел.

Таблица 17.1. Значения для ключа -C

Буква	Шестнадцатеричное число	Значение
I	0x1	Предполагается, что STDIN работает в режиме UTF-8.
O	0x2	Предполагается, что STDOUT работает в режиме UTF-8.
E	0x4	Предполагается, что STDERR работает в режиме UTF-8.
S	0x7	I + O + E
i	0x8	UTF-8 – кодировка по умолчанию в PerlIO для потоков ввода.
o	0x10	UTF-8 – кодировка по умолчанию в PerlIO для потоков вывода.
D	0x18	i + o
A	0x20	Ожидается, что элементы в @ARGV являются строками в кодировке UTF-8.
L	0x40	Обычно символы «*IOEioA*» действуют безусловно; добавление *L* делает их зависимыми от значений переменных среды, определяющих региональные настройки (LC_ALL, LC_TYPE и LANG в порядке уменьшения приоритета) – если переменные указывают на кодировку UTF-8, тогда выбирается комбинация «*IOEioA*».
a	0x100	Устанавливает ${^UTF8CACHE} в значение -1, чтобы обеспечить кэширование кода в режиме отладки в кодировке UTF-8. Вероятно, бессмысленно, если только вы не собираетесь заняться отладкой или переделкой внутренних механизмов Perl.

Например, комбинация *-COE* и ее числовой эквивалент *-C6* включают режим поддержки кодировки UTF-8 для потоков вывода STDOUT и STDERR. Повторяющиеся буквы не являются ошибкой, но и никакого дополнительного эффекта не производят.

Комбинация *io* означает, что ко всем последующим вызовам open (или аналогичным операциям ввода/вывода) в области видимости текущего файла будет неявно применяться фильтр :utf8 PerlIO. Иными словами, будет предпо-

лагаться, что входной текст имеет кодировку UTF-8, и выводиться данные будут тоже в кодировке UTF-8. Однако это всего лишь умолчания, которые можно переопределять, явно указывая фильтры в open и binmode.

Сам по себе ключ *-C* (без букв или цифр, следующих за ним) или пустая строка "" в переменной среды PERL_UNICODE имеют тот же эффект, что и комбинация *-CSDL*. Иными словами, стандартные потоки ввода/вывода и режимы открытия файлов будут поддерживать кодировку UTF-8, но только если она указана в переменных среды, определяющих региональные настройки. Такое поведение соответствует *неявному* поведению поддержки UTF-8 в v5.8.0 и в настоящее время не приветствуется.

Вы можете использовать *-C0* (или "0" в качестве значения переменной среды PERL_UNICODE), чтобы явно запретить особенности обработки Юникода, описанные выше.

Числовое значение этой настройки отражает волшебная переменная ${^UNICODE}. Она инициализируется в процессе запуска Perl, после чего становится доступной только для чтения. Если вам потребуется отменить ее действие, используйте прагму open, или функцию open с тремя аргументами, или функцию binmode с двумя аргументами.

(В версиях ниже v5.8.1 ключ *-C* был доступен только в Win32, где использовался для перехода на применение функций Win32 API с поддержкой Юникода. Эта особенность почти не использовалась на практике, поэтому данный ключ командной строки был «переориентирован».)

Примечание: начиная с версии v5.10.1, если ключ *-C* используется в строке #!, он также должен фигурировать в командной строке. Это обусловлено тем, что к моменту запуска интерпретатора Perl стандартные потоки ввода/вывода уже настроены. Для установки кодировки потоков ввода/вывода можно также использовать функцию binmode.

-d

-dt Запускает сценарий в отладчике Perl. См. главу 18. Дополнительный символ *t* указывает, что отладчик должен выполнять отладку в многопоточном окружении.

-d:MODULE[=ARG1,ARG2]

-dt:MODULE[=ARG1,ARG2]

Запускает сценарий под управлением отладочного, профилирующего или трассирующего модуля, установленного в библиотеке Perl как Devel::*MODULE*. Например, *-d:DProf* выполнит сценарий с использованием профилировщика Devel::DProf. Как и при использовании ключа *-M*, пакету Devel::*MODULE* могут передаваться параметры для интерпретации их в функции Devel::*MODULE*::import. Опять же, как и для *-M*, чтобы вместо функции import вызвать Devel::*MODULE*::unimport, используйте use *-d:-MODULE*. Список параметров, разделенных запятыми, должен начинаться символом =. Дополнительный символ *t* указывает, что отладчик используется в многопоточном окружении.

-D LETTERS

-D NUMBER

Устанавливает флаги отладки. (Работает, только если ваша версия Perl скомпилирована с поддержкой отладки, как описано ниже.) Можно задать число

NUMBER, представляющее собой сумму нужных битов, либо список букв *LETTERS*. Например, чтобы увидеть, как выполняется сценарий, используйте *-D14* или *-Dslt*. Другим полезным значением является *-D1024* или *-Dx*, выводящее скомпилированное синтаксическое дерево. Ключ *-D512* или *-Dr* выводит скомпилированные регулярные выражения. Числовое значение ключа доступно внутри сценария в специальной переменной `$^D`. Значения битов перечислены в табл. 17.2. Для простоты числа ниже приведены в шестнадцатеричном виде, но в составе ключа они должны передаваться в десятичном виде. Мы настоятельно рекомендуем использовать буквенные обозначения.

Таблица 17.2. Опции -D

Бит	Буква	Значение
0x0400000	*A*	Проверка целостности внутренних структур («Все ли чисто?»)
0x2000000	*B*	Дамп определений подпрограмм, включая специальные блоки, такие как `BEGIN`
0x0200000	*C*	Режим копирования при записи (copy-on-write)
0x0000020	*c*	Преобразования строка/число
0x0008000	*D*	Сборка мусора по завершении программы
0x0000100	*f*	Обработка форматов
0x0002000	*H*	Дамп хешей – узурпирует `values`
0x0080000	*J*	Отобразить `s,t,P-debug` в (т.е. не пропуская) операциях в пакете `DB::`
0x0000004	*l*	Обход и обработка стека контекста
0x1000000	*M*	Трассировка подстановки значений в операциях интеллектуального сопоставления
0x0000080	*m*	Распределение памяти и скалярных значений
0x0000010	*o*	Поиск методов и перегрузки
0x0000040	*P*	Вывод профилировочной информации, состояние ввода исходного файла
0x0000001	*p*	Выводить результаты лексического и синтаксического разбора (с буквой *v* выводит стек лексического разбора)
0x0800000	*q*	Подавить вывод всех сообщений `EXECUTING`
0x0040000	*R*	Включить счетчики ссылок при выводе переменных (например, при использовании *-Ds*)
0x0000200	*r*	Разбор и выполнение регулярных выражений
0x0000002	*s*	Выводить состояние стеков (с буквой *v* выводит все стеки)
0x0020000	*T*	Лексический анализ
0x0000008	*t*	Трассировка выполнения
0x0001000	*U*	Для неофициальных пользовательских забав (зарезервировано для частного использования)
0x0000800	*u*	Проверка меченых данных
0x0100000	*v*	Увеличение детализации вывода: используется в сочетании с другими флагами
0x0004000	*X*	Выделение временной памяти
0x0000400	*x*	Дамп дерева синтаксического разбора

Для всех этих флагов требуется исполняемый модуль Perl, скомпилированный с поддержкой отладки. По умолчанию Perl компилируется без поддержки отладки, поэтому вы не сможете использовать ключ *-D*, пока вы сами или ваш системный администратор не скомпилируете версию Perl для отладки. Подробности можно найти в файле *INSTALL* в каталоге с исходным кодом Perl, но вкратце суть состоит в том, чтобы передать компилятору C ключ *-DDEBUGGING* при компиляции Perl. Этот флаг автоматически устанавливается, если включить опцию *-g*, когда *Configure* запрашивает у пользователя флаги оптимизатора и отладчика.

Если вы просто хотите получить распечатку каждой строки программы Perl по ходу ее выполнения (аналогично тому, что дает *sh -x* для сценариев интерпретатора команд), ключ Perl *-D* не поможет. Вместо этого нужно сделать следующее:

```
# Синтаксис интерпретатора команд Борна (bash)
$ PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

# Синтаксис csh
% (setenv PERLDB_OPTS "NonStop=1 AutoTrace=1 frame=2"; perl -dS program)
```

За подробностями обращайтесь к главе 18.

-e PERLCODE

Может использоваться для ввода одной или более строк сценария. Если указан ключ *-e*, Perl не ищет имя файла программы в списке аргументов. Аргумент *PERLCODE* рассматривается, как если бы он заканчивался символом перевода строки, поэтому можно передать несколько ключей *-e*, чтобы создать программу из нескольких строк. (Не забудьте использовать точки с запятой, где это необходимо.) Тот факт, что *-e* добавляет в конец аргумента символ перевода строки, не значит, что вы обязаны использовать несколько ключей *-e*, – если интерпретатор команд позволяет заключать в кавычки многострочный текст, как *sh*, *ksh* и *bash*, сценарий из нескольких строк можно передать в одном аргументе *-e*:

```
$ perl -e 'print "Howdy, ";print "@ARGV!\n";' world
Howdy, world!
```

В *csh* лучше передать несколько ключей *-e*:

```
% perl -e 'print "Howdy, ";' \
       -e 'print "@ARGV!\n";' world
Howdy, world!
```

Учитываются как явные, так и неявные символы новой строки, поэтому в обоих случаях вторая команда `print` находится на строке 2 сценария *-e*.

-E PERLCODE

Действует подобно *-e*, за исключением того, что неявно включает все дополнительные возможности (в единице компиляции `main`). В версии v5.14 такими возможностями являются `say`, `state`, `switch` и `unicode_strings`. См. описание прагмы `feature` в главе 29.

-f Запрещает выполнение *$Config{sitelib}/sitecustomize.pl* при запуске.

Perl можно собрать так, что по умолчанию при запуске он будет пытаться выполнить *$Config{sitelib}/sitecustomize.pl* (в блоке `BEGIN`). Эта возможность по-

зволяет системным администраторам настраивать поведение Perl. Например, таким способом можно добавлять записи в массив @INC, чтобы Perl мог находить модули в нестандартных каталогах.

Фактически Perl вставляет следующий код:

```
BEGIN {
    do {
        local $!;
        -f "$Config{sitelib}/sitecustomize.pl";
    } && do "$Config{sitelib}/sitecustomize.pl";
}
```

Поскольку в действительности это do (а не require), модуль *sitecustomize.pl* не должен возвращать истинное значение. Код выполняется в пакете main, в собственной лексической области видимости. Но, если сценарий завершится аварийно, переменная $@ не будет установлена.

Кроме того, значение $Config{sitelib} определяется в C-коде, а не извлекается из *Config.pm*, который вообще не загружается.

Код выполняется на *очень раннем* этапе. Например, любые изменения, произведенные в @INC, можно наблюдать в результатах работы команды *perl -V*. Разумеется, соответствующие блоки END будут выполняться на самом позднем этапе.

Чтобы во время выполнения определить наличие этой возможности, можно проверить значение $Config{usesitecustomize}:

```
% perl -V:usesitecustomize
usesitecustomize='undef';
```

-F PATTERN

Задает шаблон для split, когда ключ *-a* включает авторазбивку (в противном случае не оказывает никакого действия). Шаблон может быть заключен в символы косой черты (//), двойные кавычки ("") или одинарные кавычки (''). По умолчанию он заключается в одинарные кавычки. Не забывайте, что способ передачи кавычек через интерпретатор команд зависит от используемого интерпретатора.

-h Выводит справку по ключам командной строки Perl.

-i

-i EXTENSION

Указывает, что файлы, обрабатываемые посредством конструкции <>, должны редактироваться «по месту». Это делается путем переименования входного файла, открытия выходного файла с исходным именем и выбора этого выходного файла в качестве файла по умолчанию для вызовов print, printf и write.[1] Аргумент *EXTENSION* используется для изменения имени прежнего файла с целью создания резервной копии. Если он отсутствует, резервная копия не создается и перезаписывается текущий файл. Если в *EXTENSION* нет символа *, заданная строка добавляется в конец текущего имени файла. Если *EXTENSION* содержит один или несколько символов *, каждый символ * за-

1 Технически это не совсем «по месту». Имя файла то же, но физически это другой файл.

меняется именем обрабатываемого в данный момент файла. На языке Perl это можно выразить как

```
($backup = $extension) =~ s/\*/$file_name/g;
```

Благодаря этому к имени резервной копии файла можно добавить не только суффикс, но и префикс:

```
% perl -pi'orig_*' -e 's/foo/bar/' xyx # создать копию с именем 'orig_xyx'
```

Резервные копии исходных файлов можно даже поместить в другой каталог (при условии, что он существует):

```
% perl -pi'old/*.orig' -e 's/foo/bar/' xyx # создать копию 'old/xyx.orig'
```

Следующие пары однострочных команд эквивалентны:

```
% perl -pi -e 's/foo/bar/' xyx         # перезаписать текущий файл
% perl -pi'*' -e 's/foo/bar/' xyx      # перезаписать текущий файл

% perl -pi'.orig' -e 's/foo/bar/' xyx  # сделать резервную копию в 'xyx.orig'
% perl -pi'*.orig' -e 's/foo/bar/' xyx # сделать резервную копию в 'xyx.orig'
```

Следующая команда:

```
% perl -p -i.orig -e "s/foo/bar/;"
```

эквивалентна программе:

```
#!/usr/bin/perl -pi.orig
s/foo/bar/;
```

что служит удобной сокращенной формой записи существенно более длинной программы:

```
#!/usr/bin/perl
$extension = '.orig';
LINE: while (<>) {
    if ($ARGV ne $oldargv) {
        if ($extension !~ /\*/) {
            $backup = $ARGV . $extension;
        }
        else {
            ($backup = $extension) =~ s/\*/$ARGV/g;
        }
        unless (rename($ARGV, $backup)) {
            warn "невозможно переименовать $ARGV в $backup: $!\n";
            close ARGV;
            next;
        }
        open(ARGVOUT, ">$ARGV");
        select(ARGVOUT);
        $oldargv = $ARGV;
    }
    s/foo/bar/;
}
continue {
    print; # вывод в исходное имя файла
}
select(STDOUT);
```

Этот длинный фрагмент, в сущности, идентичен простому однострочнику с ключом *-i*, за вычетом того, что в форме с ключом *-i* не требуется сравнивать $ARGV с $oldargv, чтобы узнать, что имя изменилось. Однако в нем используется ARGVOUT для выбранного дескриптора файла и восстанавливается STDOUT как дескриптор файла вывода по умолчанию после выполнения цикла. Как и в приведенном коде, Perl создает резервную копию файла независимо от того, изменился ли в действительности вывод. В описании функции eof можно найти примеры ее использования без скобок для поиска конца каждого входного файла, если требуется дописывать в конец каждого файла или сбрасывать нумерацию строк.

Если Perl не может создать резервную копию данного файла, как указано в *EXTENSION*, то выводит предупреждение и продолжает обработку остальных перечисленных файлов.

Нельзя применять ключ *-i* для создания каталогов или удаления расширений из файлов. Нельзя также использовать его с тильдой ~, чтобы обозначить домашний каталог, что и хорошо, поскольку некоторые любят включать этот символ в имена файлов резервных копий:

```
% perl -pi~ -e 's/foo/bar/' file1 file2 file3...
```

Наконец, ключ *-i* не препятствует выполнению Perl в отсутствие имен файлов в командной строке. В этом случае просто не будут созданы резервные копии, поскольку нельзя определить исходный файл, и обработка продолжится, как можно было ожидать, с чтением из STDIN и выводом в STDOUT.

-I DIRECTORY

Каталоги, задаваемые с помощью *-I*, добавляются в начало @INC, где содержатся пути поиска модулей. Как и use lib, ключ *-I* неявно добавляет каталоги, специфические для платформы. Подробности о директиве use lib приведены в главе 29.

-l
-l OCTNUM

Включает автоматическую обработку конца строки. Его применение, во-первых, приводит к удалению символа окончания строки (chomp), если указан также ключ *-n* или *-p*, а во-вторых, в $\ записывается значение *OCTNUM*, в результате чего все операторы print выводят дополнительно символ окончания строки с кодом ASCII, равным *OCTNUM*. Если *OCTNUM* опустить, то *-l* установит переменную $\ равной текущему значению $/, обычно – символу новой строки. Поэтому, чтобы ограничить строки 80 колонками, введите следующее:

```
% perl -lpe 'substr($_, 80) = ""'
```

Обратите внимание, что присваивание $\ = $/ производится при обработке ключа, поэтому разделитель входных записей может не совпадать с разделителем выходных записей, если за ключом *-l* следует ключ *-0*:

```
% gnufind / -print0 | perl -ln0e 'print "found $_" if p'
```

В этой команде $\ устанавливается равной символу новой строки, а затем $/ устанавливается равной нулевому символу. (Обратите внимание, что 0 интерпретировался бы как часть ключа *-l*, если бы следовал непосредственно за ним, поэтому мы вставили между ними ключ *-n*.)

-m и *-M*

Эти ключи загружают *MODULE*, как если бы вы выполнили use. Если задать *-MODULE* вместо *MODULE*, тогда вызывается no. Например, *-Mstrict* действует аналогично use strict, а *-M-strict* – аналогично no strict.

-m MODULE

Выполняет use *MODULE*() перед выполнением сценария.

-M MODULE

Выполняет use MODULE перед выполнением сценария. Команда образуется простой интерполяцией оставшейся части аргумента после *-M*, поэтому можно использовать кавычки, чтобы добавить код после имени модуля, например *-M'MODULE qw{foo bar}'*. Если первым символом после *-M* или *-m* является дефис (-), тогда инструкция use заменяется инструкцией no. Эту особенность можно использовать для определения минимальной допустимой версии Perl. Например, *-Mv5.14* гарантирует, что сценарий будет выполняться только под управлением Perl версии v5.14 или более поздней.

-M MODULE=ARG1, ARG2...

Маленькое синтаксическое удобство, позволяющее указать *-Mmodule=foo,bar* вместо более длинной формы *-M'module qw{foo bar}'*. Такая форма позволяет обойтись без кавычек при импортировании символов. Фактически *-Mmodule=foo,bar* генерирует следующий код:

```
use module split(/,/, q{foo,bar})
```

Важно учитывать, что в формате со знаком = нет разница между *-m* и *-M*, однако, во избежание путаницы, предпочтительнее использовать ключ *-M*.

Ключи *-M* и *-m* можно использовать только при вызове Perl из командной строки, но не как параметры в строке #!. (Если нужно вставить их в файл, почему вместо этого просто не использовать эквивалентные use или no?)

-P Был ликвидирован в Perl версии v5.12 из-за проблем с переносимостью. Используйте вместо него модуль Text::CPP из CPAN.

-n Предписывает Perl выполнять сценарий в цикле, как показано ниже, с аргументами имен файлов на манер *sed -n* и *awk*:

```
LINE:
while (<>) {
    ...  # здесь вызывается сценарий
}
```

Метку LINE можно использовать в сценарии, несмотря на то что фактически она отсутствует в файле.

Обратите внимание: по умолчанию строки не выводятся. Если требуется вывод строк, обратитесь к описанию ключа *-p*. Вот эффективный способ удалить все файлы, созданные больше недели назад:

```
find . -mtime +7 -print | perl -nle unlink
```

Эта команда выполняется быстрее, чем при использовании ключа *-exec* в *find*(1), поскольку не приходится запускать процесс для каждого найденного файла. Этот прием работает некорректно при наличии символов перевода строки в путях к файлам, что можно исправить, как показано в примере из описа-

ния ключа *-0*. Поразительное совпадение: блоки BEGIN и END позволяют организовать перехват управления до или после неявного цикла, совсем как в *awk*.

-p Предписывает Perl выполнять сценарий в цикле, как показано ниже, с аргументами имен файлов на манер *sed*:

```
LINE:
while (<>) {
    ... # здесь вызывается сценарий
}
continue {
    print || die "-p destination: $!\n";
}
```

Метку LINE можно использовать в сценарии, несмотря на то что фактически она отсутствует в файле.

Если по какой-либо причине файл, заданный в аргументе, нельзя открыть, Perl выведет предупреждение и перейдет к следующему файлу. Обратите внимание, что строки выводятся автоматически. Ошибка, возникшая во время вывода, считается фатальной. Еще одно поразительное совпадение: блоки BEGIN и END можно использовать для перехвата управления до или после неявного цикла, совсем как в *awk*.

-s Разрешает элементарный анализ ключей, находящихся в командной строке после имени сценария, но предшествующих аргументам с именами файлов или ключу «--», завершающему обработку ключей. Все найденные ключи удаляются из @ARGV, зато создаются переменные, имена которых совпадают с именами ключей. Совмещение ключей в этом случае не разрешается, поскольку -s позволяет использовать многосимвольные имена ключей.

Следующий сценарий выведет "true", только если будет запущен с ключом *-foo*.

```
#!/usr/bin/perl -s
if ($foo) { print "true\n" }
```

Если ключ имеет вид *-xxx=yyy*, переменная $xxx устанавливается равной значению аргумента, следующего за знаком равенства (в данном случае "yyy"). Следующий сценарий выведет "true", только если будет запущен с ключом *-foo=bar*.

```
#!/usr/bin/perl -s
if ($foo eq 'bar') { print "true\n" }
```

Обратите внимание, что такой ключ, как *--help* создаст переменную ${-help}, не совместимую со strict refs. Кроме того, использование этого ключа в сценариях, где разрешен вывод предупреждений, может привести к появлению фиктивных предупреждений «used only once» (используется только один раз).

-S Заставляет Perl использовать при поиске сценария переменную среды PATH (только если имя сценария не содержит символа-разделителя каталогов).

Обычно этот ключ применяется, чтобы упростить эмуляцию запуска посредством #! на платформах, не поддерживающих #!. На многих платформах, имеющих интерпретатор команд, совместимый с интерпретатором Борна или C Shell, можно использовать такую команду:

```
#!/usr/bin/perl
eval "exec /usr/bin/perl -S $0 $*"
        if $running_under_some_shell;
```

Система игнорирует первую строку и передает сценарий интерпретатору */bin/sh*, который пытается выполнить сценарий Perl как сценарий интерпретатора команд. Интерпретатор команд выполняет вторую строку как обычную команду системы и тем самым запускает интерпретатор Perl. В некоторых системах $0 может не содержать полный путь, поэтому *-S* сообщает Perl, что при необходимости следует найти сценарий. После того как Perl найдет сценарий, он выполнит синтаксический анализ строк и проигнорирует их, поскольку переменная `$running_under_some_shell` всегда имеет ложное значение. Более удачной заменой $* может стать конструкция ${1+"$@"}, обрабатывающая пробелы в именах файлов пробелы и другие специальные символы, но она не работает, если сценарий интерпретируется как *csh*-сценарий. Чтобы запустить *sh*, а не *csh*, некоторые системы должны заменить строку #! строкой, содержащей просто двоеточие, которую Perl вежливо проигнорирует. Другие системы не могут это контролировать, так что требуют совершенно неимоверной конструкции, которая будет работать как в *csh*, *sh*, так и в *perl*, например такой:

```
eval '(exit $?0)' && eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
    & eval 'exec /usr/bin/perl -S $0 $argv:q'
                   if 0;
```

Да, сие безобразно[1], как и системы, устроенные подобным образом.

На некоторых платформах ключ *-S* также заставляет Perl добавлять суффиксы к искомому имени файла. Например, на платформах Win32 добавляются суффиксы *.bat* и *.cmd*, когда поиск исходного имени оказывается неудачным, а имя не оканчивается одним из этих суффиксов. Если Perl скомпилирован с поддержкой отладки, с помощью ключа Perl *-Dp* можно посмотреть, как происходит поиск.

Если переданное имя файла содержит символ-разделитель каталогов (даже если путь относительный, а не абсолютный), и файл найти не удается, платформы, неявно добавляющие расширения имен файлов (не UNIX), сделают это и будут поочередно искать файлы с этими добавленными расширениями.

На платформах, схожих с DOS, если сценарий не содержит разделителей имен каталогов, его поиск осуществляется сначала в текущем каталоге, а потом в путях из переменной PATH. На платформах UNIX поиск сценария осуществляется строго в списке каталогов PATH из соображений безопасности, требующих предотвращения случайного запуска программы из текущего каталога, когда явного запроса на это нет.

-t Действует подобно ключу -T, но проверка меченых данных генерирует предупреждения, а не фатальные ошибки. Эти предупреждения можно контролировать обычным способом с помощью `no warnings qw(taint)`.

Примечание: *этот ключ не служит заменой -T*! Он предназначен *исключительно* как временная мера на этапе разработки, для подстраховки устаревшего кода: для действующего кода и для новых программ, создаваемых с нуля, всегда следует использовать *-T*.

---

[1] Термин употреблен нами после тщательного рассмотрения.

-T Включает проверки «меченых» данных («taint» checks), которые становится возможным использовать. Обычно эти проверки осуществляются только при выполнении сценария с установленными битами setuid или setgid. Неплохо явно включить их для программ, выполняемых от имени другого пользователя, таких как программы CGI. См. главу 20.

Из соображений безопасности Perl должен увидеть этот ключ как можно раньше; обычно это значит, что ключ следует расположить в начале командной строки или строки `#!`. Если он расположен недостаточно близко к началу, Perl выразит неудовольствие.

-u Заставляет Perl выполнить дамп памяти после компиляции сценария. Теоретически этот дамп можно превратить в выполняемый модуль с помощью программы *undump* (не входит в комплект поставки Perl). При этом запуск ускоряется ценой некоторого перерасхода дискового пространства (который можно сократить путем усечения – stripping – выполняемого модуля). Если вы хотите сначала выполнить часть сценария, а потом – дамп, используйте вместо этого ключа оператор Perl `dump`. Примечание: наличие программы *undump* зависит от платформы; в некоторых версиях Perl она может отсутствовать. Она постепенно вытесняется новым транслятором кода из Perl в C, который отличается лучшей переносимостью (но пока является экспериментальным).

-U Позволяет Perl выполнять небезопасные операции. В данное время «небезопасными» считаются только операции удаления каталогов при работе с правами суперпользователя и запуск программ с setuid, для которых непройденные проверки меченых данных не являются фатальными, а лишь приводят к выдаче предупреждений. Учтите, что вывод предупреждений должен быть включен, чтобы предупреждения о меченых данных действительно выводились.

-v Выводит номер версии и уровень патчей выполняемого модуля Perl, а также некоторые дополнительные сведения.

-V Выводит список основных параметров конфигурации Perl и текущее значение `@INC`.

-V:NAME

Выводит в `STDOUT` значение указанной переменной конфигурации. *NAME* может содержать символы регулярных выражений, такие как «.» для сопоставления любому символу и «.*» для сопоставления произвольной последовательности символов.

```
% perl -V:man.dir
man1dir='/usr/local/man/man1'
man3dir='/usr/local/man/man3'

% perl -V:'.*threads'
d_oldpthreads='undef'
use5005threads='define'
useithreads='undef'
usethreads='define'
```

При попытке запросить несуществующую переменную конфигурации, ее значение сообщается как `"UNKNOWN"`. Внутри программы сведения о конфигурации можно получить с помощью модуля `Config`, однако он не поддерживает шаблоны в индексах хешей:

```
% perl -MConfig -le 'print $Config{man1dir}'
/usr/local/man/man1
```

См. описание модуля Config.

-w Выводит предупреждения о переменных, встречающихся лишь один раз, и скалярных величинах, используемых до присваивания им значений. Также выводятся предупреждения о переопределении подпрограмм и ссылках на неопределенные дескрипторы файлов или дескрипторы, открытые только для чтения, но используемые для записи. Выводятся также предупреждения об использовании в числовом контексте величин, не похожих на числа, об использовании массива в скалярном контексте, о рекурсии с глубиной выше 100 и о массе других вещей. Все эти сообщения отмечены символами «(W)» в *perldiag*.

Этот ключ просто устанавливает глобальную переменную $^W. Он не оказывает влияния на лексические предупреждения – см. описание ключей *-W* и *-X*. Включать или отключать отдельные сообщения можно с помощью прагмы warnings (или no warnings), описанной в главе 29.

-W Включает безусловный вывод всех предупреждений в программе, даже если они отключены локально с помощью no warnings или $^W = 0. Распространяется на все файлы, загруженные с помощью use, require или do. Можете рассматривать этот ключ как Perl-эквивалент команды *lint*(1).

-x

-x DIRECTORY

Предписывает Perl извлечь сценарий, встроенный в сообщение. Весь мусор, предшествующий сценарию, отбрасывается, а началом считается строка, начинающаяся с #! и содержащая слово "perl". Все поддающиеся интерпретации ключи в строке после слова "perl" будут применены. Если задано имя каталога, Perl перейдет в него перед запуском сценария. Ключ *-x* способен удалить мусор только в начале сообщения, но не в конце. Сценарий должен заканчиваться меткой __END__ или __DATA__, если в конце есть мусор, который нужно проигнорировать. (При желании сценарий может обработать весь мусор в конце сообщения или его часть с помощью дескриптора файла DATA. Теоретически этот дескриптор позволяет даже найти начало файла при помощи *seek* и обработать мусор, предшествующий сообщению.)

-X Безоговорочно и до конца выполнения отключает вывод всех предупреждений – в точности обратно тому, что делает флаг *-W*.

Переменные среды

В дополнение к различным ключам, явно изменяющим режим работы Perl, можно устанавливать переменные среды, влияющие на различные основные режимы. Способ установки переменных среды зависит от системы, однако есть один прием, о котором нужно знать, если вы используете *sh*, *ksh* или *bash*: можно временно устанавливать переменную среды для одной команды, как если бы она была своеобразным ключом. Эта переменная должна быть установлена перед командой:

```
$ PATH='/bin:/usr/bin' perl myproggie
```

Нечто похожее можно делать в порожденных интерпретаторах для *csh* и *tcsh*:

```
% (setenv PATH "/bin:/usr/bin"; perl myproggie)
```

В других случаях переменные среды обычно устанавливаются в некотором файле с именем типа *.chsrc* или *.profile*, находящемся в домашнем каталоге пользователя. В *csh* и *tcsh* можно сказать:

```
% setenv PATH '/bin:/usr/bin'
```

А в *sh*, *ksh* и *bash* команда будет выглядеть так:

```
$ PATH='/bin:/usr/bin'; export PATH
```

В других системах есть свои способы установки этих переменных на полупостоянной основе. Вот несколько переменных среды, которые распознает Perl:

`HOME`

Используется, когда `chdir` вызывается без аргумента.

`LC_ALL, LC_CTYPE, LC_COLLATE, LC_NUMERIC, PERL_BADLANG`

Переменные среды, управляющие обработкой языком Perl данных, характерных для отдельных естественных языков. См. страницу *perllocale* справочного руководства.

`LOGDIR`

К этой переменной Perl обращается, когда `chdir` вызывается без аргументов, но переменная `HOME` не установлена.

`PATH`

Используется при запуске подпроцессов и для поиска программ, если указан ключ *-S*.

`PERL5DB`

Команда предназначена для загрузки отладчика. По умолчанию:

```
BEGIN { require 'perl5db.pl' }
```

Дополнительные сведения об использовании этой переменной см. в главе 18.

`PERL5DB_THREADED`

Если эта переменная имеет истинное значение, отладчик будет выполнять отладку в многопоточном окружении.

`PERL_ALLOW_NON_IFS_LSP` *(только в версии для Win32)*

Если имеет значение 1, разрешает использовать LSP (Layered Service Provider – многоуровневый поставщик услуг), несовместимый с IFS (Installable File System – разрешенная к установке файловая система). Обычно Perl пытается отыскать IFS-совместимый LSP, необходимый для интерпретации Windows-сокетов как действительных дескрипторов файлов. Однако это может вызывать проблемы при наличии брандмауэров, таких как *McAfee Guardian*, требующих, чтобы все приложения использовали его LSP, не совместимый с IFS, из-за чего Perl обычно старается не использовать такие LSP.

Запись значения 1 в эту переменную среды требует, чтобы Perl просто использовал первый подходящий LSP, имеющийся в каталоге, удовлетворив требования *McAfee Guardian* (и в этом конкретном случае Perl тоже сохранит работоспособность, потому что LSP брандмауэра *McAfee Guardian* в действительно-

сти предпринимает дополнительные усилия, чтобы обеспечить работоспособность приложений, требующих совместимости с IFS).

PERL_DEBUG_MSTATS

Применяется, только если Perl скомпилирован с функцией malloc (т. е. если *perl -V:d_mymalloc* возвращает "define"). Если переменная установлена, то после выполнения возвращается статистика использования памяти. Если переменная установлена равной целому числу, превышающему единицу, статистика распределения памяти выводится и после компиляции.

PERL_DESTRUCT_LEVEL

Применяется, только если Perl скомпилирован с поддержкой отладки, и управляет режимом глобального удаления объектов и других ссылок. Дополнительную информацию можно найти в разделе «PERL_DESTRUCT_LEVEL» на странице *perlhacktips* (*http://perldoc.perl.org/perlhacktips.html*).

PERL_DL_NONLAZY

Установите эту переменную в значение 1, чтобы заставить Perl искать *все* неопределенные символы во время загрузки динамической библиотеки. По умолчанию поиск символов производится при первой попытке их использования. Настройка этой переменной может пригодиться при тестировании расширений, поскольку гарантирует появление сообщений об ошибках при наличии опечаток в именах функций, даже если эти функции не вызываются тестами.

PERL_ENCODING

Не используйте эту переменную. Она опирается на неработоспособную прагму encoding.

PERL_HASH_SEED

(Начиная с v5.8.1.) Используется для рандомизации внутренней хеш-функции Perl. Для имитации поведения, «предшествовавшего версии 5.8.1», присвойте этой переменной целое число (ноль соответствует поведению v5.8.0). Фраза «предшествовавшего версии 5.8.1» означает, кроме всего прочего, что ключи хешей всегда будут генерироваться в том же порядке при каждом запуске Perl.

По умолчанию большинство хешей возвращает элементы в том же порядке, что и в Perl версии v5.8.0. Если при вставке ключа в любой хеш обнаруживаются патологические данные, этот хеш переключается на альтернативную случайную последовательность хеш-ключей.

Рандомизация выполняется по умолчанию, если переменная PERL_HASH_SEED не установлена. Если Perl скомпилирован с флагом *-DUSE_HASH_SEED_EXPLICIT*, рандомизация по умолчанию не выполняется, если переменная PERL_HASH_SEED не установлена.

Если переменная PERL_HASH_SEED не установлена или имеет нечисловое значение, для определения начального значения последовательности псевдослучайных чисел используется генератор псевдослучайных чисел, предоставляемый операционной системой и библиотеками.

Имейте в виду, что начальное значение последовательности псевдослучайных чисел является критичной информацией с точки зрения безопасности. Рандомизация хешей выполняется с целью защититься от локальных и удаленных атак на код Perl. При установке начального значения последовательно-

сти псевдослучайных чисел вручную эта защита может быть частично или полностью нарушена. За дополнительными сведениями обращайтесь к разделу «Algorithmic Complexity Attacks» в *perlsec* (*http://perldoc.perl.org/perlsec.html*) и к разделу «ENVIRONMENT» в *perlrun* (*http://perldoc.perl.org/perlrun .html*).

`PERL_HASH_SEED_DEBUG`

(Начиная с v5.8.1.) Установите значение 1 в этой переменной для вывода (в `STDERR`) начального значения последовательности псевдослучайных чисел для хеш-функции при запуске сценария. Эта переменная в паре с переменной `PERL_HASH_SEED` [см. раздел «PERL_HASH_SEED» в *perlrun* (*http://perldoc.perl.org/perlrun.html*)] предназначена для отладки случайных ошибок, обусловленных рандомизацией хеш-функции.

Имейте в виду, что начальное значение последовательности псевдослучайных чисел является критичной информацией с точки зрения безопасности. Зная это значение, можно реализовать атаку на код Perl даже удаленно. За дополнительными сведениями обращайтесь к разделу «Algorithmic Complexity Attacks» в *perlsec* (*http://perldoc.perl.org/perlsec.html*). *Не раскрывайте это значение тем, кто не должен его знать.* См. также описание функции `hash_seed()` в модуле `Hash::Util`.

`PERL_MEM_LOG`

Если компиляция вашей версии Perl выполнялась с флагом `-Accflags=-DPERL_MEM_LOG`, установкой переменной среды `PERL_MEM_LOG` можно включить ведение журнала отладочных сообщений. Значение переменной имеет вид *number*`[m][s][t]`, где *number* – это номер дескриптора файла, куда следует выполнять запись (2, по умолчанию), а комбинация букв определяет необходимость вывода информации об операциях с памятью (m)emory и/или скалярными значениями (s)v, а также информацию о текущем времени (t)imestamp. Например, `PERL_MEM_LOG=1mst` обеспечит вывод всей информации в `STDOUT`. Можно также указать дескриптор любого другого открытого файла:

```
bash$ 3>foo3 PERL_MEM_LOG=3m perl
```

`PERL_ROOT` *(только в версии для VMS)*

Логическое устройство и путь к корневому каталогу установки Perl для включения в `@INC`, только для ОС VMS. К переменным среды, оказывающим влияние на Perl, в системе VMS также относятся `PERLSHR`, `PERL_ENV_TABLES` и `SYS$TIMEZONE_DIFFERENTIAL`, но все они являются необязательными, и их описание можно найти в *perlvms*, а также в файле *README.vms*, входящем в дистрибутив с исходными текстами Perl.

`PERL_SIGNALS`

Для Perl версии v5.8.1 и более поздних. Если установлена в значение `unsafe`, обработка сигналов выполняется (немедленно, но небезопасно) как в версиях Perl до 5.8.0. Если установлена в значение `safe`, используется алгоритм безопасной (отложенной) обработки сигналов. См. раздел «Deferred Signals (Safe Signals)» в *perlipc* (*http://perldoc.perl.org/perlipc.html*).

`PERL5SHELL` *(только в версии для Microsoft)*

Может устанавливать альтернативный командный процессор (интерпретатор команд), который Perl должен использовать внутренне для выполнения ко-

манд через обратные апострофы или вызовы system. По умолчанию это *cmd.exe /x/c* в WinNT и *command.com /c* в Win95. Предполагается, что в качестве разделителя в значении выступают пробелы. Каждому символу, который должен быть экранирован (такому, как пробел или обратная косая черта), должна предшествовать обратная косая черта.

Обратите внимание, что Perl не использует в этих целях переменную COMSPEC, поскольку для нее пользователи находят самые разные применения, что вызывает проблемы переносимости между платформами. Кроме того, если Perl будет работать с интерпретатором команд, непригодным для интерактивной работы, установка такого интерпретатора в COMSPEC может помешать правильному функционированию других программ (которые обычно ищут в COMSPEC интерпретатор, пригодный для интерактивного использования).

PERL5LIB

Список каталогов, разделенных двоеточиями[1], где следует искать библиотечные файлы Perl, прежде чем заглядывать в стандартную библиотеку и в текущий каталог. Если в указанных каталогах существуют специфические для данной архитектуры каталоги, они автоматически включаются в путь поиска. Для обратной совместимости с более старыми версиями при неопределенной переменной PERL5LIB осуществляется обращение к PERLLIB.

При проверке меченых данных (если программа запущена с битами setuid или setgid, либо с ключом *-T*) переменные, определяющие порядок поиска библиотек, не используются. В такие программы следует включать директиву use lib.

PERL5OPT

Ключи командной строки по умолчанию. Ключи из этой переменной используются, как если бы они присутствовали в каждой командной строке Perl. Допустимы только ключи *-[DIMUdmw]*. При проверке меченых данных (если программа запущена с битами setuid или setgid, либо с ключом *-T*) эта переменная игнорируется. Если PERL5OPT начинается с *-T*, включается проверка меченых данных, а все последующие ключи игнорируются.

PERLIO

Список фильтров PerlIO, разделенных пробелами (или двоеточиями). Если при сборке подсистема ввода/вывода Perl была настроена на использование PerlIO, эти фильтры оказывают влияние на операции ввода/вывода Perl.

Имена фильтров обычно начинаются двоеточием (например, :perlio), чтобы подчеркнуть их сходство с «атрибутами» переменных. Но код, выполняющий анализ строк с определениями фильтров (который также используется для анализа переменной среды PERLIO), интерпретирует двоеточия как разделители.

Если переменная PERLIO не определена или содержит пустое значение, используется набор фильтров по умолчанию для текущей платформы. Например, :unix:perlio – в UNIX-подобных системах, и :unix:crlf – в Windows и в других DOS-образных системах.

Список определяет фильтры по умолчанию для *всех* операций ввода/вывода. По этой причине в списке могут указываться только встроенные фильтры, поскольку внешние фильтры, такие как :encoding(*LAYER*), должны загружаться

---

1 В UNIX и UNIX-подобных ОС, а в Microsoft в качестве разделителя – точка с запятой.

отдельно перед их использованием. См. описание прагмы open в главе 29, где рассказывается, как обеспечить возможность использования внешних фильтров по умолчанию.

Возможно, кому-то потребуется включить некоторые фильтры в переменную среды PERLIO, поэтому ниже приводится их краткое описание.

:bytes Псевдофильтр, *отключающий* действие флага :utf8 для нижележащего фильтра. Маловероятно, что встретится в глобальной переменной среды PERLIO сам по себе обычно применяется в сочетаниях :crlf:bytes и :perlio:bytes.

:crlf Фильтр, преобразующий CRLF в "\n". Различает «текстовые» и «двоичные» файлы по аналогии с MS-DOS и похожими операционными системами. (В настоящее время вся имитация поведения MS-DOS сводится лишь к распознаванию символа Control-Z как признака конца файла.)

:mmap Фильтр, реализующий «чтение» файлов с применением mmap, помещая файл (целиком) в адресное пространство процесса, а затем используя его в качестве «буфера» PerlIO.

:perlio Повторная реализация «STDIO-подобного» механизма буферизации в виде «фильтра» PerlIO. Фактически, *:perlio* обращается к нижележащим фильтрам для выполнения своих операций (обычно :unix).

:pop Экспериментальный псевдофильтр, удаляющий самый верхний фильтр. Обращайтесь с ним так же осторожно, как с нитроглицерином.

:raw Псевдофильтр, управляющий другими фильтрами. Применение фильтра raw эквивалентно вызову binmode($fh). Он заставляет поток передавать каждый байт как есть, без какого-либо декодирования. В частности, запрещает преобразование CRLF и отменяет действие флага :utf8.

В отличие от ранних версий Perl, сейчас :raw ***не ограничивается тем,*** чтобы инвертировать действие флага :crlf. Действие других фильтров, которые могут оказывать влияние на двоичную природу потока, также отменяется.

:stdio Этот фильтр реализует интерфейс PerlIO, обертывая вызовы стандартной библиотеки ввода/вывода ANSI C. Обеспечивает не только ввод/вывод, но и буферизацию. Обратите внимание, что фильтр :stdio не выполняет преобразование CRLF, даже если это является обычным поведением для платформы. Для этого следует явно использовать фильтр :crlf, располагающийся выше.

:unix Самый нижний фильтр, вызывающий функции read, write, lseek и другие.

:utf8 Псевдофильтр, позволяющий флагу уровнем ниже сообщить Perl, что вывод должен производиться в кодировке UTF-8, а ввод должен интерпретироваться как текст в кодировке UTF-8. Он не выполняет проверку корректности кодировки, поэтому входные данные должны обрабатываться с особой осторожностью. Используя этот фильтр для ввода, всегда включайте (желательно фатальные) предупреждения, связанные с проверкой кодировки UTF-8. Или используйте :encoding(UTF-8) при чтении данных в кодировке UTF-8.

:win32 На платформах Win32 этот *экспериментальный* фильтр использует «родные» для этой платформы «дескрипторы» системы ввода/вывода вместо UNIX-подобных числовых дескрипторов файлов. Известно, что в версии v5.14 этот фильтр содержит некоторые ошибки.

Набор фильтров, устанавливаемых по умолчанию, должен давать приемлемые результаты на всех платформах.

Для UNIX и UNIX-подобных платформ это будет эквивалент «unix perlio» или «stdio». Если системная библиотека обеспечивает быстрый доступ к буферу, при настройке предпочтение отдается реализации «stdio», в противном случае используется реализация «unix perlio».

На платформах Win32 версия v5.14 по умолчанию использует «unix crlf». Реализация «stdio» в версии для Win32 имеет несколько ошибок или, мягко говоря, недостатков, в зависимости от того, какой компилятор C использовался для сборки Perl IO. Применение нашего собственного фильтра crlf в качестве буфера позволяет избежать этих проблем и обеспечить большее единообразие. Фильтр crlf, наряду с буферизацией, осуществляет преобразование CRLF в "\n".

В качестве самого нижнего на платформе Win32 в Perl v5.14 используется фильтр unix, и, как следствие, C-подпрограммы, манипулирующие числовыми дескрипторами файлов. Существует экспериментальный фильтр win32, который, как ожидается, будет доработан в будущем и станет фильтром по умолчанию для платформ Win32.

Переменная среды PERLIO полностью игнорируется, когда Perl действует в режиме меченных данных.

PERLIO_DEBUG

Если содержит имя файла или устройства, некоторые операции подсистемы PerlIO будут отражены в этом файле-журнале, открытом в режиме дописывания. В UNIX обычно используется следующим образом:

```
% env PERLIO_DEBUG=/dev/tty perl script ..
```

Примерный эквивалент в Win32:

```
> set PERLIO_DEBUG=CON
> perl script
```

Эта функция отключается в setuid-сценариях и в сценариях, запускаемых с флагом *-T*.

PERLLIB

Список каталогов, разделенных двоеточиями, где следует искать библиотечные файлы Perl, прежде чем заглядывать в стандартную библиотеку и в текущий каталог. Если определена PERL5LIB, то PERLLIB не используется.

PERL_UNICODE

Эквивалент ключа *-C* командной строки. Имейте в виду, что это не логическая переменная. Запись в нее значения "1" – это неправильный способ «включить поддержку Юникода» (чтобы ни подразумевалось под этим). Однако вы можете использовать "0", чтобы «отключить поддержку Юникода» (или просто удалить переменную PERL_UNICODE в интерпретаторе команд перед запуском Perl).

При работе с текстом наиболее универсальным значением для этой переменной является "AS": оно обеспечивает неявное декодирование @ARGV из кодировки UTF-8 и вызов binmode для всех трех стандартных потоков — STDIN, STDOUT и STDERR — с фильтром :utf8. Используйте это значение, если предполагается, что текст — не просто поток байтов, но последовательность символов в кодировке UTF-8. В некоторых случаях еще более полезным может оказаться значение "ASD", но оно приводит также к изменению кодировки по умолчанию для всех дескрипторов файлов с двоичной на :utf8, что может нарушить работу многих старых программ, которые предполагают, что работают с двоичными (или текстовыми, если говорить о Windows) потоками, и потому не утруждают себя вызовом binmode. Это особенно характерно для программ UNIX. Поэтому литеру "D" здесь лучше использовать избирательно, для запуска отдельных программ.

Встроенный фильтр :utf8 по умолчанию не возбуждает исключения и даже не выводит предупреждения, столкнувшись с некорректными данными в кодировке UTF-8 при вводе, поэтому, чтобы обеспечить корректное поведение при использовании фильтра :utf8 для потоков ввода, необходимо также включить предупреждения "utf8". Для этого можно указать флаг *-Mwarnings=utf8* в командной строке, чтобы включить предупреждения, и *-Mwarnings=FATAL,utf8*, чтобы включить исключения. Это соответствует инструкциям use warnings "utf8" и use warnings FATAL => "utf8" внутри программы. См. также раздел «Получение данных в Юникоде» в главе 6.

SYS$LOGIN *(в версии для VMS)*

Используется, если chdir вызывается без аргументов, а переменные HOME и LOGDIR не установлены.

Сам Perl не использует никакие другие переменные среды, кроме тех, которые должен сделать доступными выполняемой программе или ее порожденным процессам. Что до модулей, стандартных или пользовательских, они могут работать с любыми переменными среды. Например, прагма re использует PERL_RE_TC и PERL_RE_COLORS, модуль Cwd использует PWD, а модуль CGI обращается к многочисленным переменным среды, устанавливаемым демоном HTTP (т.е. веб-сервером) для передачи сценарию CGI.

Программы, запущенные с битом setuid, поступят правильно, если прежде чем что-либо делать, выполнят следующие строки (чтобы все было по-честному):

```
$ENV{PATH}  = "/bin:/usr/bin";    # или что еще нужно
$ENV{SHELL} = "/bin/sh" if exists $ENV{SHELL};
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

Подробности читайте в главе 20.

18

Отладчик Perl

Прежде всего, пытались ли вы пользоваться директивой warnings?

Если вызвать Perl с ключом *-d*, программа будет запущена под управлением отладчика Perl. Отладчик – это интерактивная среда Perl, выводящая приглашения для ввода команд. Команды отладчика позволяют просматривать исходный код, устанавливать точки останова, выводить дамп стека вызова функции, изменять значения переменных и т. д. Любая команда, не распознанная отладчиком, выполняется (при помощи eval) как код Perl внутри отлаживаемого в данный момент программного пакета. (Чтобы не мешать отлаживаемой программе, информацию о собственном состоянии отладчик хранит в пакете DB.) Это настолько удобно, что часто отладчик запускают, только чтобы интерактивно проверить действие конструкций Perl. В таком случае неважно, какая программа отлаживается, поэтому выберем не особенно содержательную:

```
% perl -de 42
```

Отладчик Perl не является отдельной программой, как это обычно бывает в типичной среде программирования. Флаг *-d* сообщает компилятору о необходимости вставить исходный код в дерево синтаксического разбора, который он будет передавать интерпретатору. Это значит, что код должен быть правильно скомпилирован, прежде чем с ним сможет работать отладчик. Если компиляция успешна, интерпретатор загрузит особый библиотечный файл, содержащий собственно отладчик.

```
% perl -d /path/to/program
```

Программа остановится непосредственно перед первой инструкцией этапа выполнения (об инструкциях этапа компиляции рассказывается в разделе «Использование отладчика» ниже) и предложит ввести команду отладчика. Когда отладчик останавливается, то показывает строку кода, которую *собирается* выполнить, а не только что выполненную.

Встретив очередную строку, отладчик сначала проверит наличие точки останова, выведет ее (если находится в режиме трассировки), выполнит действия, определяемые командой a (как описывается далее в разделе «Команды отладчика»), и,

наконец, выведет командное приглашение, если строка является точкой останова или работа выполняется в пошаговом режиме. В противном случае он выполнит строку обычным образом и перейдет к следующей.

Использование отладчика

Командное приглашение отладчика выглядит примерно так:

```
DB<8>
```

или так:

```
DB<<17>>
```

где число показывает количество выполненных команд отладчика. Механизм журнала команд в стиле *csh* позволяет повторно вызывать команды по номеру. Например, !17 повторит команду с номером 17. Количество угловых скобок указывает на глубину вложенности отладки. Например, более одной пары скобок можно увидеть, если, находясь в точке останова, попытаться вывести результат вызова функции, в которой также присутствует точка останова.

Чтобы ввести в отладчик инструкцию, состоящую из нескольких строк, например определение подпрограммы с несколькими операторами, нужно перед переводом строки, которым обычно завершаются команды отладчика, поместить обратную косую черту. Например:

```
DB<1> for (1..3) {      \
    cont:   print "ok\n"; \
    cont: }
    ok
    ok
    ok
```

Допустим, вы хотите запустить отладчик со своей программкой (назовем ее *camel_flea*[1]) и остановиться, добравшись до функции с именем infested. Вот как это нужно сделать:

```
% perl -d camel_flea
  Loading DB routines from perl5db.pl version 1 07
  Editor support available

  Enter h or 'h h' for help, or 'man perldebug' for more help.

  main::(camel_flea:2): pests('bactrian', 4);
DB<1>
```

Отладчик остановит программу прямо перед первой инструкцией этапа выполнения (об инструкциях этапа компиляции рассказывается ниже) и предложит ввести команду. Еще раз напомним, что когда отладчик останавливается, он показывает строку, которую *собирается* выполнить, а не ту, которая только что выполнена. Строка может выглядеть не совсем так, как в исходном файле, особенно если текст обработан препроцессором.

[1] Camel flea (англ.) – верблюжьи блохи. – *Прим. перев.*

Итак, нужно остановиться, как только программа дойдет до функции `infested`, поэтому создадим точку останова:

```
DB<1> b infested
DB<2> c
```

Теперь отладчик продолжит выполнение, пока не наткнется на эту функцию, и тогда сообщит следующее:

```
main::infested(camel_flea:8):     my $bugs = int rand(3);
```

Чтобы увидеть «окно» исходного кода, окружающего точку останова, воспользуйтесь командой `w`:

```
DB<2> w
5     }
6
7         sub infested {
8==>b         my $bugs = int rand(3);
9:            our $Master;
10:           contaminate($Master);
11:           warn "needs wash"
12                if $Master && $Master->isa("Human");
13
14:           print "got $bugs\n";

DB<2>
```

Как показывает маркер `==>`, текущая строка имеет номер 8, а символ `b` указывает, что строка является точкой останова. Определи мы действие для этой точки останова, в строке также присутствовал бы символ `a`. Точками останова могут быть только строки, номера которых сопровождаются двоеточиями.

Чтобы увидеть последовательность вызовов, попросите отладчик вывести дамп стека командой `T`:

```
DB<2> T
$ = main::infested called from file 'Ambulation.pm' line 4
@ = Ambulation::legs(1, 2, 3, 4) called from file 'camel_flea' line 5
. = main::pests('bactrian', 4) called from file 'camel_flea' line 2
```

Первый символ (`$`, `@`, или `.`) указывает, в каком контексте была вызвана функция – скалярном, списочном или пустом соответственно. Мы видим три строки потому, что на момент вызова вывода стека находились на глубине трех функций. Ниже объясняется значение каждой из трех строк:

- Первая строка говорит, что вы находились в функции `main::infested`, когда запустили трассировку стека. Она сообщает, что функция была вызвана в скалярном контексте из строки 4 файла *Ambulation.pm*. Она показывает также, что функция была вызвана без каких-либо аргументов, т.е. как `&infested` вместо обычного `infested()`.
- Во второй строке показано, что функция `Ambulation::legs` была вызвана в списочном контексте из строки 5 файла *camel_flea* с четырьмя указанными аргументами.
- В третьей строке показано, что `main::pests` была вызвана в пустом контексте из строки 2 файла *camel_flea*.

Если имеются инструкции, выполняемые на этапе компиляции, такие как код в блоках BEGIN и CHECK или директивы use, то обычно отладчик в них *не* останавливается, в отличие от require и блоков INIT, поскольку последние относятся к этапу выполнения (см. главу 16). Инструкции этапа компиляции можно трассировать, если установить опцию AutoTrace в PERLDB_OPTS.

Существуют некоторые возможности управления отладчиком Perl из самой программы. Например, программа может автоматически создать точку останова в некоторой подпрограмме, если запущена под отладчиком. Из кода на Perl можно также передать управление отладчику, выполнив следующую инструкцию, которая безвредна, если отладчик не запущен:

```
$DB::single = 1
```

Если установить переменную $DB::single равной 2, получится эквивалент команды n, в то время как значение 1 эмулирует команду s. Чтобы имитировать команду t, нужно установить переменную $DB::trace в значение 1.

Другим способом отладки модуля является установка точки останова на *загрузку*:

```
DB<7> b load c:/perl/lib/Carp.pm
Will stop on load of `c:/perl/lib/Carp.pm'.
```

с последующим перезапуском отладчика командой R. Более точное прицеливание осуществляет команда b compile subname, позволяющая остановиться сразу после компиляции заданной подпрограммы.

Команды отладчика

При вводе команд в отладчике не обязательно завершать их точкой с запятой. Используйте обратную косую черту для продолжения строк (но только находясь в отладчике).

Поскольку отладчик выполняет инструкции с помощью eval, значения my и local уничтожаются после возврата из команды. Если команда отладчика совпадает с именем какой-либо функции в вашей собственной программе, просто добавьте перед вызовом функции какой-либо символ, который не позволит спутать его с командой отладчика, например или +.

Если объем вывода встроенной команды отладчика превосходит количество строк на экране, предварите имя команды символом конвейеризации, и вывод будет производиться в постраничном формате:

```
DB<1> |h
```

У отладчика есть масса команд, которые мы подразделяем (достаточно произвольно) на команды пошаговой обработки и прогона, работы с точками останова, трассировки, вывода, поиска кода, автоматического выполнения инструкций и, разумеется, «прочие».

Вероятно, самой важной командой является h – команда вывода справки. Если в ответ на приглашение отладчика ввести h h, вы получите компактный вариант справки, умещающийся на одном экране. Если ввести h *COMMAND*, вы получите справку по указанной команде отладчика.

Пошаговая обработка и прогон

Отладчик действует, *пошагово* выполняя программу строка за строкой. Описанные ниже команды позволяют управлять тем, какие строки при пошаговом выполнении отладчик будет пропускать, а на каких останавливаться.

s [*EXPR*]

Команда отладчика s выполняет программу в пошаговом режиме. Это значит, что отладчик выполняет очередную строку программы, пока не достигнет следующей инструкции, при необходимости выполняя вход в подпрограммы. Если очередная строка содержит вызов функции, отладчик останавливается на первой строке в этой функции. Если команда сопровождается выражением *EXPR*, содержащим вызовы функций, последние также выполняются пошагово.

n [*EXPR*]

Команда n выполняет вызовы подпрограмм, не входя в них, и останавливается на следующей инструкции на том же уровне (или более высоком). Если команда сопровождается выражением *EXPR*, содержащим вызовы функций, они будут выполняться с остановками перед каждой инструкцией.

<ENTER>

Если просто нажать клавишу Enter в ответ на приглашение отладчика, будет повторно выполнена предыдущая команда n или s.

Команда . возвращает внутренний указатель отладчика на последнюю выполненную строку и выводит ее.

r Эта команда продолжает выполнение до возврата из текущей выполняющейся подпрограммы. Она выводит возвращаемое значение, если установлен параметр PrintRet, что происходит по умолчанию.

Точки останова

b
b *LINE*
b *CONDITION*
b *LINE CONDITION*
b *SUBNAME*
b *SUBNAME CONDITION*
b postpone *SUBNAME*
b postpone *SUBNAME CONDITION*
b compile *SUBNAME*
b load *FILENAME*

Команда отладчика b устанавливает *точку останова* перед строкой с номером *LINE*, приказывая отладчику остановить выполнение программы в этой точке и предоставить вам возможность покопаться в данных. Если номер строки *LINE* опущен, точка останова создается в строке, которая должна выполняться следующей. Если задано условие *CONDITION*, оно проверяется каждый раз перед выполнением инструкции: остановка происходит, только если значение *CONDITION* истинно. Точки останова можно создавать только в начальных строках выполняемых инструкций. Заметьте, что в условиях не используется оператор if:

```
b 237 $x > 30
b 237 ++$count237 < 11
b 33 /pattern/i
```

Команда b *SUBNAME* создает точку останова (возможно, условную) перед первой строкой указанной подпрограммы. *SUBNAME* может быть переменной, содержащей ссылку на код, в этом случае *CONDITION* не поддерживается.

Существует также несколько способов создания точек останова в коде, который еще даже не скомпилирован. Команда b postpone создает точку останова (возможно, условную) перед первой строкой подпрограммы *SUBNAME* после того, как она будет скомпилирована.

Команда b compile создает точку останова на первой инструкции, которая должна быть выполнена после компиляции *SUBNAME*. Заметьте, что в отличие от формы b postpone, эта команда создает точку останова вне рассматриваемой подпрограммы, на инструкции, которая будет выполнена после компиляции подпрограммы, а не после ее вызова.

Форма b load создает точку останова в первой выполняемой строке файла. Аргумент *FILENAME* должен представлять полный путь к файлу, как в значениях %INC.

d
d *LINE*

Удаляет точку останова в строке с номером *LINE*, а если номер строки опущен, удаляется точка останова в строке, которая должна быть выполнена следующей.

D Удаляет все точки останова.

L Выводит список всех точек останова и действий.

c
c *LINE*

Продолжает выполнение, дополнительно предоставляя возможность установить одноразовую точку останова в строке с номером *LINE*.

Трассировка

T Выводит стек вызовов.

t
t *EXPR*

Переключает режим трассировки так, что выводится каждая выполняемая строка программы. См. также описание параметра AutoTrace далее в этой главе. Если задано выражение *EXPR*, отладчик трассирует его выполнение. См. также раздел «Автоматическое выполнение» далее в этой главе.

W
W *EXPR*

Добавляет *EXPR* в список глобальных *контрольных выражений* (*watch expressions*). Контрольным называется выражение, вызывающее останов при изменении его значения. Если *EXPR* не задано, все контрольные выражения удаляются.

Вывод данных

Отладчик Perl имеет несколько команд для исследования структур данных в момент, когда программа прерывается в точке останова.

p
p *EXPR*

Эта команда оказывает такое же действие, как `print DB::OUT` *EXPR* в текущем пакете. В частности, поскольку это лишь собственная функция `print`, принадлежащая Perl, вложенные структуры данных и объекты не отображаются, а для их вывода следует использовать команду `x`. Обработчик `DB::OUT` осуществляет вывод на терминал (или в окно редактора) независимо от того, куда переадресован стандартный вывод.

x
x *EXPR*

Команда `x` вычисляет свое выражение в списочном контексте и выводит результат в «красивом» виде. Это означает, что вложенные структуры выводятся рекурсивно при надлежащей перекодировке непечатаемых символов.

V
V *PKG*
V *PKG VARS*

Эта команда выводит все (или только перечисленные в списке *VARS*) переменные в указанном пакете *PKG* (или, по умолчанию, пакете `main`), используя «красивую» печать. В хешах выводятся ключи и значения, управляющие символы выводятся наглядно, встроенные структуры выводятся в удобочитаемом виде и т.д. Своим действием эта команда напоминает вызов команды `x` для каждой переменной, за исключением того, что `x` работает и с лексическими переменными. Кроме того, идентификаторы здесь следует задавать *без* указания типа, т.е. опуская символы вроде `$` или `@`. Например:

```
V Pet::Camel SPOT FIDO
```

Вместо имен переменных в *VARS* можно использовать шаблон *~PATTERN* или *!PATTERN*, чтобы вывести переменные, имена которых соответствуют (или не соответствуют) заданному шаблону.

X
X *VARS*

Эта команда представляет собой то же, что и `V` *CURRENTPACKAGE*, где *CURRENTPACKAGE* является пакетом, в котором была скомпилирована текущая строка.

H
H *-NUMBER*

Эта команда выводит заданное количество (*NUMBER*) последних команд. В журнале запоминаются только команды длиннее одного символа (иначе большинство из них было бы `s` или `n`). Если аргумент *NUMBER* опущен, выводятся все команды.

Поиск кода

С помощью этих команд можно находить и выводить участки программы.

l
l *LINE*
l *SUBNAME*
l *MIN+INCR*
l *MIN-MAX*

Команда l выводит несколько следующих строк программы, или строку с номером *LINE*, или несколько первых строк подпрограммы *SUBNAME*, или указанный фрагмент кода.

Команда l *MIN+INCR* выведет *INCR*+1 строк, начиная с *MIN*. Команда l *MIN-MAX* выведет строки с номерами с *MIN* по *MAX*.

\- Эта команда выводит несколько предыдущих строк программы.

w
w [*LINE*]

Выводит окно (несколько строк), окружающее строку с номером *LINE* (если указан), или текущую строку, если номер строки *LINE* опущен.

f *FILENAME*

Позволяет просмотреть другую программу или выражение eval. Если *FILENAME* не является полным именем пути, который можно найти в значениях %INC, то интерпретируется как регулярное выражение для поиска нужного вам файла.

/*PATTERN*/

Осуществляет поиск по шаблону *PATTERN* в коде программы в прямом направлении; замыкающий символ / не обязателен. Шаблон *PATTERN* можно опустить, в этом случае повторяется предыдущий поиск.

?*PATTERN*?

Осуществляет поиск по шаблону *PATTERN* в коде программы в обратном направлении; замыкающий символ ? не обязателен. Если *PATTERN* опущен, повторяется предыдущий поиск.

S
S *PATTERN*
S !*PATTERN*

Команда S выводит имена подпрограмм, соответствующие (или, в случае использования префикса !, не соответствующие) шаблону *PATTERN*. Если шаблон *PATTERN* опущен, перечисляются все подпрограммы.

Действия и выполнение команд

Находясь в отладчике, можно задать действия, которые должны быть выполнены в определенные моменты. Можно также запускать внешние программы.

a
a *COMMAND*
a *LINE*
a *LINE COMMAND*

Эта команда устанавливает действие, которое должно быть выполнено перед выполнением строки с номером *LINE* или текущей строки, если номер строки опущен. Например, следующая команда приводит к выводу $foo всякий раз, когда программа достигает строки 53:

```
a 53 print "DB FOUND $foo\n"
```

Если параметр *COMMAND* не задан, действие для строки *LINE* удаляется. Если не заданы ни *LINE*, ни *COMMAND*, удаляется действие для текущей строки.

A Команда A отладчика удаляет все действия.

<
< ?
< *EXPR*
<< *EXPR*

В команде < *EXPR* задается выражение Perl, которое должно вычисляться перед каждым выводом приглашения отладчика. Можно добавить еще одно выражение, используя команду << *EXPR*, вывести список выражений командой < ? и удалить все выражения простой командой <.

>
> ?
> *EXPR*
>> *EXPR*

Команды > выполняются подобно родственным командам <, но после вывода приглашения отладчика, а не перед тем.

{
{ ?
{ *COMMAND*
{{ *COMMAND*

Команды отладчика { выполняются аналогично <, но задают не выражение на языке Perl, а команду отладчика, которую нужно выполнить перед выводом приглашения. Если вместо нее ошибочно введен блок кода, выводится предупреждение. Если это как раз то, что вам требуется, введите ;{ ... } или даже do { ... }.

!
! *NUMBER*
! *-NUMBER*
! *PATTERN*

Одиночный ! повторяет предыдущую команду. Аргумент *NUMBER* определяет, какую команду из буфера нужно выполнить, например ! 3 выполнит третью команду, введенную в отладчике. Если аргументу *NUMBER* предшествует символ «-», команды отсчитываются в обратном направлении: ! -3 выполнит третью от конца команду. Если вместо числа задать шаблон *PATTERN* (без символов косой черты), выполнится последняя команда, начало которой совпадает с шаблоном. См. также параметр отладчика recallCommand.

!! *CMD*

Эта команда отладчика выполняет в подпроцессе внешнюю команду *CMD*, которая будет читать данные из DB::IN и писать в DB::OUT. См. также параметр отладчика shellBang. Эта команда использует интерпретатор команд, указанный в $ENV{SHELL}, что иногда может мешать правильной интерпретации статуса, сигнала и данных дампа памяти. Если нужно, чтобы команда возвращала правильное значение, установите переменную $ENV{SHELL} в значение /bin/sh.

|
|*DBCMD*
||*PERLCMD*

Команда |*DBCMD* выполняет команду отладчика *DBCMD*, передавая вывод DB::OUT программе $ENV{PAGER}. Она часто используется с командами, которые выводят большой объем информации, например:

```
DB<1> |V main
```

Заметьте, что это относится к командам отладчика, а не к командам, вводимым из интерпретатора команд. Если вы хотите передать результат выполнения команды *who* программе постраничного просмотра, можно сделать так:

```
DB<1> !!who | more
```

Команда ||*PERLCMD* аналогична |*DBCMD*, но, кроме того, временно изменяет DB::OUT вызовом select, поэтому вывод всех вызовов print, printf или write без дескриптора файла также будет передаваться по конвейеру. Например, если некая функция генерирует большой объем информации вызовом print, для постраничного вывода данных вместо предыдущей следует использовать такую команду:

```
DB<1> sub saywho { print "Users: ", `who` }
DB<2> ||saywho()
```

Прочие команды

q и ^D

С помощью этих команд осуществляется выход из отладчика. Это предпочтительный способ выхода, хотя иногда действует дважды повторенная команда exit. Установите параметр inhibit_exit в значение 0, если хотите остаться в отладчике по окончании прогона программы. Может также потребоваться установить $DB::finished в 0, если вы хотите проследить процесс уничтожения глобальных объектов в пошаговом режиме.

R Перезапускает отладчик и начинает новый сеанс. Отладчик пытается сохранить преемственность работы между различными сеансами, однако некоторые настройки и параметры командной строки могут быть утеряны. В настоящее время сохраняются следующие настройки: журнал команд, точки останова, действия, параметры отладчика и параметры командной строки Perl *-w*, *-I* и *-e*.

=
= *ALIAS*
= *ALIAS VALUE*

Эта команда выводит текущее значение псевдонима *ALIAS*, если не задано значение *VALUE*. Если оно задано, то определяет новую команду отладчика с именем *ALIAS*. Если опущены оба параметра, *ALIAS* и *VALUE*, выводится список всех текущих псевдонимов, например:

```
= quit q
```

ALIAS должен быть простым идентификатором и транслироваться также в простой идентификатор. Более сложные псевдонимы можно создавать

путем добавления собственных записей непосредственно в `%DB::aliases`**. См. раздел «Настройка отладчика» далее в этой главе.**

`man`
`man` *`MANPAGE`*

Вызывает программу по умолчанию для просмотра документации с указанной страницей или, если аргумент *`MANPAGE`* **опущен, просто запускает эту программу. Если этим средством просмотра является программа *man*, для ее вызова используются данные из текущего** `%Config`**. При необходимости автоматически добавляется префикс «**`perl`**», что позволяет вводить в отладчике** `man debug` **и** `man op`**.**

В системах, где утилита *man* отсутствует, отладчик вызывает *perldoc*; чтобы изменить этот режим, установите в `$DB::doccmd` **имя средства просмотра, которое вам необходимо. Это можно сделать с помощью файла *rc* или непосредственным присваиванием.**

`O`
`O` *`OPTION ...`*
`O` *`OPTION? ...`*
`O` *`OPTION=VALUE...`*

Команда `O` **позволяет манипулировать параметрами настройки отладчика, перечисленными в разделе «Параметры настройки отладчика» далее в этой главе. Команда** `O` *`OPTION`* **устанавливает в значение 1 каждый из перечисленных параметров** *`OPTION`*. **Если за** *`OPTION`* **следует вопросительный знак, выводится текущее значение параметра.**

Команда `O` *`OPTION=VALUE`* **устанавливает значение параметра; если значение** *`VALUE`* **содержит пробельный символ, его следует заключить в кавычки. Например, можно задать** `O pager="less -MQeicsNfr"`**, чтобы *less* использовалась с этими конкретными флагами. Кавычки могут быть либо одинарными, либо двойными, но при этом нужно использовать управляющие символы для внедрения в строку того типа кавычек, в которые она заключена. Необходимо также экранировать любую обратную косую черту, непосредственно предшествующую кавычке, но предназначенную для экранирования этой кавычки. Иными словами, выполняйте правила заключения в одинарные кавычки независимо от того, какой тип кавычек используется на практике. В ответ на команду установки параметра отладчик выводит его значение, всегда используя форму записи в одинарных кавычках:**

```
DB<1> O OPTION='this isn\'t bad'
              OPTION = 'this isn\'t bad'

DB<2> O  OPTION="She said, \"Isn't it?\""
              OPTION = 'She said, "Isn\'t it?"'
```

В силу исторических причин присваивание *`=VALUE`* **необязательно, но значение, по умолчанию равное единице, устанавливаются только там, где это допустимо, т.е. по большей части в логических параметрах. Конкретные значения** *`VALUE`* **лучше присваивать с помощью** `=`**. Имена параметров** *`OPTION`* **можно сокращать, но делать это, пожалуй, следует, только если вы стремитесь к загадочности и непрозрачности. Одновременно можно устанавливать несколько параметров. Их список приведен в разделе «Параметры настройки отладчика».**

Настройка отладчика

Способов настройки отладчика существует предостаточно, поэтому вам вряд ли потребуется модифицировать его. Находясь в самом отладчике, можно изменить его режим с помощью команды O, из командной строки это можно сделать через переменную среды PERLDB_OPTS и путем выполнения предварительно заданных команд, хранимых в файлах *rc*.

Поддержка отладки в редакторах

Механизм журнала команд отладчика не позволяет производить редактирование в командной строке, как это делают многие интерпретаторы команд: нельзя извлекать предыдущие строки с помощью ^p[1] или перемещаться к началу строки с помощью ^a[2], хотя можно выполнять предыдущие строки, используя синтаксис с восклицательным знаком, знакомый пользователям системного интерпретатора команд. Однако если установить модули Term::ReadKey и Term::ReadLine из CPAN, можно получить полные возможности редактирования, аналогичные имеющимся в GNU-утилите *readline*(3).

Если у вас установлен *emacs*, можно организовать его взаимодействие с отладчиком Perl для создания интегрированной среды разработки, напоминающей интеграцию *emacs* с отладчиками C. В состав Perl входит файл настройки, позволяющий модифицировать *emacs* так, чтобы он работал как структурный редактор, частично распознающий синтаксис Perl. Этот файл находится в каталоге *emacs* дистрибутива исходного кода Perl. Пользователям *vi* следует также присмотреться к *vim* (и его мышино-оконной версии *gvim*), позволяющему выделять цветом ключевые слова Perl.

Существует также аналогичная по возможностям надстройка за авторством Тома Кристиансена для взаимодействия с редактором *vi* любого крупного дистрибутива и оконной системой X11. Надстройка действует аналогично интегрированной многооконной среде, предоставляемой *emacs*, в которой отладчик управляет редактором. На момент написания данной книги оставалось неясным, войдет ли она в итоге в дистрибутив Perl, и где будет располагаться, но мы решили сообщить вам о такой возможности.

Настройка с помощью файлов инициализации

Некоторые настройки можно произвести, внеся изменения в код файла инициализации *.perldb* или *perldb.ini* (имя файла зависит от операционной системы). Такой файл инициализации содержит код Perl, а не команды отладчика, и он обрабатывается раньше, чем анализируется переменная среды PERLDB_OPTS. Например, можно следующим образом создавать псевдонимы, добавляя элементы в хеш %DB::alias:

```
$alias{len}  = 's/^len(.*)/p length($1)/';
$alias{stop} = 's/^stop (at|in)/b/';
$alias{ps}   = 's/^ps\b/p scalar /';
$alias{quit} = 's/^quit(\s*)/exit/';
$alias{help} = 's/^help\s*$/|h/';
```

[1] Комбинация клавиш Ctrl-P. – *Прим. перев.*

[2] Комбинация клавиш Ctrl-A. – *Прим. перев.*

Используя внутренний API отладчика, можно изменять параметры настройки вызовами функции в файле инициализации:

```
parse_options("NonStop=1 LineInfo=db.out AutoTrace=1 frame=2"),
```

Если в файле инициализации определена подпрограмма afterinit, она будет вызвана по окончании инициализации отладчика. Файл инициализации может находиться в текущем или в домашнем каталоге. Поскольку он может содержать произвольные инструкции Perl, по соображениям безопасности его владельцем должен быть суперпользователь (root) или текущий пользователь, а запись в него должна быть разрешена только его владельцу.

Если потребуется модифицировать отладчик, скопируйте *perl5db.pl* из библиотеки Perl под новым именем и делайте с ним все, что вашей душе угодно. При этом вам потребуется установить переменную среды PERL5DB примерно так:

```
BEGIN { require "myperl5db.pl" }
```

В крайнем случае переменную PERL5DB можно также использовать для настройки отладчика, непосредственно устанавливая внутренние переменные или вызывая внутренние функции отладчика. Учтите, однако, что все переменные и функции, не документированные здесь или в электронных страницах руководства *perldebug*, *perldebguts* или DB, предназначены только для внутреннего употребления и могут быть изменены без предупреждения.

Параметры настройки отладчика

Отладчик имеет множество параметров настройки, которые можно установить с помощью команды O — интерактивно, через переменные среды или файл инициализации.

recallCommand, ShellBang

Символы, используемые для повторного вызова команды или вызова системного интерпретатора команд. По умолчанию тот и другой устанавливаются равными !.

pager

Программа, используемая для постраничного вывода в командах, начинающихся символом |. По умолчанию используется $ENV{PAGER}. Поскольку отладчик руководствуется текущими настройками терминала для вывода полужирного шрифта и воспроизведения эффекта подчеркивания, вывод результата работы некоторых команд отладчика через программу постраничного вывода может оказаться нечитаемым, если она не пропускает escape-последовательности в неизменном виде.

tkRunning

Запуск под управлением модуля Tk с выводом приглашения (посредством ReadLine).

signalLevel, warnLevel, dieLevel

Уровень детализации сообщений. По умолчанию отладчик не касается обработки исключительных ситуаций и предупреждений, поскольку их изменение может нарушить работу правильно выполняющихся программ.

Чтобы отключить этот безопасный режим по умолчанию, присвойте этим параметрам положительные значения. На уровне 1 можно отслеживать возникновение всех предупреждений (это часто раздражает) или исключительных ситуаций (это часто ценно). К несчастью, отладчик не различает фатальные и нефатальные исключительные ситуации. Если dieLevel имеет значение 1, нефатальные исключительные ситуации тоже трассируются и бесцеремонно изменяются, если возникают при выполнении eval для ваших строк или в модулях, которые вы пытаетесь загрузить. Если dieLevel имеет значение 2, отладчик не интересуется, откуда возникают исключительные ситуации: он захватывает право обработки исключительной ситуации и выводит трассировочную информацию, а затем модифицирует все исключительные ситуации, украшая их собственными деталями. Это, возможно, и полезно в некоторых задачах трассировки, но, скорее всего, безнадежно запутает любую программу, которая серьезно относится к обработке исключений.

Отладчик постарается вывести сообщение при поступлении неперехваченных сигналов INT, BUS или SEGV. Если вы находитесь в медленном системном вызове (таком как wait, accept или read, выполняя операцию с клавиатурой или сокетом) и не установили собственный обработчик сигналов $SIG{INT}, то не сможете вернуться в отладчик с помощью Control-C, поскольку обработчик сигнала $SIG{INT} в отладчике не поймет, что должен возбудить исключительную ситуацию, чтобы выполнить *longjmp*(3) из медленного системного вызова.

AutoTrace

Устанавливает режим трассировки (аналогично команде t, но может располагаться в PERLDB_OPTS).

LineInfo

Назначает файл или конвейер для вывода сведений о номерах строк. Если это конвейер (например, |visual_perl_db), используется короткое сообщение. Именно этот механизм применяется для взаимодействия с подчиненным редактором или визуальным отладчиком – скажем, при использовании специальных обработчиков (hooks) для *vi* или *emacs* или графического отладчика *ddd*.

inhibit_exit

Значение 0 разрешает выход по окончании отладки программы.

PrintRet

Если параметр установлен (по умолчанию – да), значение, возвращаемое командой r, выводится.

ornaments

Управляет внешним видом командной строки (см. электронную документацию по Term::ReadLine). В настоящее время нет способа отключить украшения (ornaments), из-за чего вывод на некоторых мониторах или с некоторыми программами постраничного просмотра бывает нечитаемым. Считается дефектом программы.

frame

Управляет выводом сообщений при входе в подпрограммы и выходе из них. Если выражение frame & 2 ложно, сообщения выводятся только при входе. (Вывод при выходе может быть полезен, если он перемежается другими сообщениями.)

Если выражение frame & 4 **истинно, выводятся аргументы функций вместе с данными о контексте и вызвавшей программе. Если истинно выражение** frame & 8, **для выводимых аргументов включаются перегруженная** stringify **и обработка функцией** tie FETCH. **Если истинно выражение** frame & 16, **выводится значение, возвращаемое подпрограммой.**

Максимальная длина списка аргументов при выводе определяется следующим параметром.

maxTraceLen

Максимальная длина списка аргументов при установленном бите 4[1] **в параметре** frame.

Следующие параметры управляют поведением команд V, X **и** x:

arrayDepth, hashDepth

Определяют максимальное количество элементов массива и хеша для вывода, соответственно. Если не определены, выводятся все элементы.

compactDump, veryCompact

Определяют стиль вывода массивов и хешей. Если включен параметр compactDump, **короткие массивы могут выводиться в одной строке.**

globPrint

Вывод содержимого таблицы типов данных typeglobs.

DumpDBFiles

Вывод массивов, содержащих отлаживаемые файлы.

DumpPackages

Вывод таблиц символов пакетов.

DumpReused

Вывод содержимого «повторно используемых» адресов.

quote, HighBit, undefPrint

Определяют стили вывода строк. Значением по умолчанию для quote **является** auto; **можно установить формат с двойными или одинарными кавычками, записав в параметр или ' соответственно. По умолчанию символы с установленным старшим битом выводятся без цитирования.**

UsageOnly

Когда этот параметр включен, вместо содержимого переменных пакета выводятся элементарные дампы расходования памяти для каждого пакета на основании суммарного размера строк, обнаруженных в переменных пакета. Поскольку используется таблица символов пакета, лексические переменные игнорируются.

Автоматическое выполнение

В переменной $ENV{PERLDB_OPTS} **можно определить начальные значения параметров инициализации** TTY, noTTY, ReadLine **и** NonStop.

---

[1] **Здесь имеется в виду истинность выражения** frame & 4, **т.е. в действительности проверяется второй бит целого числа (отсчет начинается с нуля).** – ***Прим. перев.***

Если в файле инициализации содержится строка:

```
parse_options("NonStop=1 LineInfo=tperl.out AutoTrace"),
```

программа работает без вмешательства пользователя, а данные трассировки помещаются в файл *tperl.out*. (Если вы будете ее прерывать, следует задать для `LineInfo` значение */dev/tty*, если хотите что-то увидеть.)

Следующие параметры можно задать только при запуске. В файле инициализации их можно установить вызовом `parse_options("OPT=VAL")`.

`TTY`

Терминал, используемый для ввода/вывода при отладке.

`noTTY`

Если этот параметр установлен, отладчик входит в режим `NonStop` и не подключается к терминалу. При прерывании (или явной передаче управления отладчику через установку переменных `$DB::signal` или `$DB::single` в программе Perl) отладчик подключается к терминалу, указанному при запуске в параметре `TTY` или выбранному вами во время выполнения с помощью модуля `Term::Rendezvous`.

В этом модуле должен быть реализован метод с именем `new`, возвращающий объект с двумя методами: `IN` и `OUT`. Они должны возвращать дескрипторы файлов для использования отладчиком при вводе и выводе соответственно. Метод `new` должен проверить аргумент, содержащий значение `$ENV{PERLDB_NOTTY}` при запуске, либо `"$ENV{HOME}/.perldbtty$$"`. Этот файл не проверяется на правильность владения им или защищенность от записи, поэтому теоретически существует угроза безопасности.

`ReadLine`

Ложное значение отключает поддержку `ReadLine` в отладчике, что позволяет отлаживать приложения, которые сами используют модуль `ReadLine`.

`NonStop`

Если этот параметр установлен, отладчик переходит в неинтерактивный режим, пока не будет прерван или пока ваша программа не установит `$DB::signal` или `$DB::single`.

Иногда параметры можно однозначно определить, обозначив их лишь первой буквой, но мы советуем всегда указывать их имена полностью для лучшей читаемости и обеспечения совместимости в будущем.

Вот пример использования переменной среды `PERLDB_OPTS` для автоматической установки параметров.[1] При этих значениях параметров программа запускается в неинтерактивном режиме с выводом информации о каждом входе в подпрограмму и для каждой выполняемой строки. Выходные данные трассировки помещаются в файл *tperl.out*. Это позволяет программе использовать обычные устройства ввода/вывода и избежать смешивания своих данных с данными трассировки.

```
$ PERLDB_OPTS="NonStop frame=1 AutoTrace LineInfo=tperl.out" perl -d myprog
```

---

[1] Здесь мы применили синтаксис интерпретатора команд *sh* для демонстрации установки переменных среды. Пользователи других интерпретаторов должны внести соответствующие изменения.

Если вы прервали выполнение программы, следует быстро установить O LineInfo=/dev/tty или другое устройство, существующее на вашей платформе, иначе вы не увидите приглашения отладчика.

Поддержка отладчика

Perl предоставляет специальные функции для отладки на этапе компиляции и на этапе выполнения, чтобы создать такую же среду отладки, как для стандартного отладчика. Эти функции не следует путать с ключами *perl -D*, которые можно использовать, только если Perl скомпилирован с ключом *-DDEBUGGING*.

Например, при каждом обращении к встроенной функции caller из пакета DB, аргументы, помещенные в соответствующий кадр стека, копируются в массив @DB::args. При вызове Perl с ключом *-d* включаются следующие дополнительные функции:

- Perl вставляет перед первой строкой программы значение $ENV{PERL5DB} (или BEGIN {require 'perl5db.pl'}, если такой переменной нет).
- В массив @{"_<$filename"} помещаются строки $filename для всех файлов, скомпилированных Perl. То же касается строк, выполняемых через eval и содержащих подпрограммы, или выполняемых в данное время. Значения $filename для строк в eval имеют вид (eval 34). Утверждения в регулярных выражениях имеют вид (re_eval 19).
- В хеш %{"_<$filename"} помещаются точки останова и действия. В качестве ключей выступают номера строк. Вместо целых хешей можно определять отдельные элементы. Perl интересует здесь только логическая истинность, хотя значения, используемые *perl5db.pl*, имеют вид "$break_condition\0$action". Значения в этом хеше имеют волшебное (magic) свойство в числовом контексте: они равны нулю, если на строке нельзя создать точку останова.

  То же касается eval-вычисляемых строк, в которых содержатся подпрограммы, или выполняемых в данный момент. Значения $filename для строк, выполняемых через eval, имеют вид (eval 34) или (re_eval 19).
- В скаляре ${"_<$filename"} содержится "_<$filename". То же относится к eval-вычисляемым строкам, содержащим подпрограммы, или выполняемым в данный момент. Значения $filename для строк, выполняемых через eval, имеют вид (eval 34) или (re_eval 19).
- После компиляции каждого файла, загружаемого через require, но до его выполнения, вызывается DB::postponed(*{"_<$filename"}), если существует подпрограмма DB::postponed. В данном случае $filename является расширенным именем файла в соответствии со значениями в %INC.
- После компиляции всех подпрограмм *subname* проверяется $DB::postponed{*subname*}. Если такой ключ и подпрограмма DB::postponed существуют, выполняется вызов DB::postponed(*subname*).
- Поддерживается хеш %DB::sub, ключами которого являются имена подпрограмм, а значения имеют вид *filename:startline-endline*. *filename* имеет вид (eval 34) для подпрограмм, определенных внутри eval, или (re_eval 19) для подпрограмм в утверждениях внутри регулярных выражений.
- Когда выполнение программы доходит до места, где может содержаться точка останова, вызывается подпрограмма DB::DB, если одна из переменных –

$DB::trace, $DB::single или $DB::signal — имеет истинное значение. Эти переменные не локализуются. Эта функция выключается, когда выполнение происходит внутри подпрограммы DB::DB, в том числе для вызываемых из нее функций, если только выражение $^D & (1<<30) не является истинным.

- Когда выполнение программы достигает точки вызова подпрограммы, вместо нее вызывается &DB::sub(*args*), при этом $DB::sub содержит имя вызываемой подпрограммы. Этого не происходит, если подпрограмма компилировалась в пакете DB.

Обратите внимание: если для работы &DB::sub требуются внешние данные, до ее завершения невозможен вызов каких-либо подпрограмм. Для стандартного отладчика пример такой зависимости дает переменная $DB::deep (на какое количество уровней рекурсии можно пройти вглубь отладчика до обязательного выхода).

Создание собственного отладчика

Минимальный работающий отладчик состоит из одной строки:

```
sub DB::DB {}
```

которую, хотя она абсолютно ничего не делает, легко можно определить через переменную среды PERL5DB:

```
$ PERL5DB="sub DB::DB {}" perl -d your-program
```

Другой компактный, но более полезный отладчик можно создать так:

```
sub DB::DB {print ++$i; scalar <STDIN>}
```

Этот небольшой отладчик выводит порядковый номер каждой встретившейся инструкции и ждет нажатия клавиши Enter для продолжения работы.

Следующий отладчик, как ни мал он с виду, вполне функционален:

```
{
    package DB;
    sub DB {}
    sub sub {print ++$i, " $sub\n"; &$sub}
}
```

Он выводит порядковый номер вызова подпрограммы, а также ее вызываемой подпрограммы. Заметьте, что &DB::sub должна компилироваться в пакете DB, как мы здесь это сделали.

Если базой для вашего нового отладчика является текущий отладчик, существует несколько обработчиков, которые помогут его настроить. При запуске отладчик читает файл начальной конфигурации из текущего каталога или домашнего каталога. После чтения файла отладчик читает переменную среды PERLDB_OPTS и выполняет ее анализ как остатка строки 0 ..., которая могла быть введена в ответ на приглашение отладчика.

Отладчик поддерживает также «волшебные» внутренние переменные, такие как @DB::dbline, %DB::dbline, служащие псевдонимами для @{"::_<*current_file*"} %{"::_<*current_file*"}. Здесь *current_file* является файлом, выбранным в данный момент либо явно с помощью команды отладчика f, либо неявно в соответствии с логикой выполнения программы.

Настройку облегчают некоторые функции. DB::parse_options(*STRING*) выполняет анализ строки как параметра команды O. DB::dump_trace(*SKIP* [,*COUNT*]) пропускает заданное количество кадров стека и возвращает список, содержащий сведения об указанном количестве кадров стека вызова (или обо всех кадрах, если *COUNT* отсутствует). Каждый элемент списка является ссылкой на хеш с ключами "context" (., $ или @), "sub" (имя подпрограммы или сведения о eval), "args" (undef или ссылка на массив), "file" и "line". Функция DB::print_trace(*FH*, *SKIP* [,*COUNT* [,*SHORT*]]) выводит в форматированном виде сведения о кадрах стека вызовов в заданный файловый дескриптор. Две последние функции удобно использовать как аргументы команд отладчика < и <<.

Нет необходимости все это заучивать. На деле, когда требуется отладить программу, мы обычно добавляем в код некоторое количество команд вывода и повторно прогоняем программу.

А на пике формы мы еще вспоминаем, что хорошо бы включить вывод предупреждений. Этого часто оказывается достаточно, чтобы обнаружить проблему и не рвать на голове волосы (или то, что от них осталось). Однако если этого недостаточно, приятно сознавать, что за ключом *-d* в терпеливом ожидании находится чудный отладчик, который может сделать почти все, *кроме* того, чтобы найти за вас ошибку.

Самое важное, что следует запомнить о настройке отладчика, это, пожалуй, следующее: не ограничивайте свое представление о программных ошибках (bugs) как о чем-то, что приводит Perl в расстройство. Если программа приводит в расстройство *вас*, это тоже ошибка. Выше мы показали пару очень простых пользовательских отладчиков. В следующем разделе мы рассмотрим пользовательский отладчик иного сорта, который (бывает) в состоянии помочь отладить ошибку под названием «Эта штука собирается когда-нибудь закончить работу?».

Профилировщик Perl

На момент написания этих строк в состав Perl входил профилировщик с названием Devel::DProf. Однако, когда вы будете читать эти строки, возможно, он уже канет в Лету. Версия Perl v5.16, которая должна выйти, когда эта книга окажется на полках книжных магазинов, уже не будет включать этот старый профилировщик. Большинство пользователей профилировщиков перейдет на другой инструмент, Devel::NYTProf. Мы расскажем вам о профилировщике Devel::DProf, поскольку он пока еще входит в состав Perl, но мы также расскажем и о новом профилировщике, который пока отсутствует в Perl.

Эти профилировщики достаточно тяжеловесны, но помимо них существуют и другие. В архиве CPAN доступен также профилировщик Devel::SmallProf, сообщающий о времени, потраченном на исполнение каждой строки программы. Он поможет выявить использование дорогостоящих (в смысле времени выполнения) конструкций Perl. Большинство встроенных функций отличается высокой эффективностью, но очень легко написать регулярное выражение, время выполнения которого увеличивается в геометрической прогрессии с увеличением объема входных данных. См. также раздел «Эффективность» в главе 21, где приводятся дополнительные советы.

Devel::DProf

Хотите, чтобы ваша программа работала быстрее? Ну, конечно, да. Но прежде следует спросить себя: «Действительно ли нужно тратить время, чтобы заставить программу работать быстрее?» Оптимизация на досуге может доставлять удовольствие,[1] но обычно есть более привлекательные способы провести время. Иногда следует заранее составить план и запустить программу во время перерыва на кофе (или использовать запуск в качестве повода для перерыва на кофе.) Но если действительно требуется, чтобы ваша программа работала быстрее, начать следует с ее профилирования. Профилировщик может сообщить, какие участки программы дольше всего выполняются, чтобы вам не пришлось терять время, совершенствуя подпрограмму, оказывающую незначительное влияние на общее время выполнения.

В состав Perl входит профилировщик `Devel::DProf`. Его можно использовать для профилирования программы Perl в *mycode.pl*, введя:

```
% perl -d:DProf mycode.pl
```

Хотя мы и назвали `DProf` профилировщиком, поскольку именно в этом состоит его функция, в нем используется тот же самый механизм, который мы обсуждали ранее в этой главе. `DProf` является просто отладчиком, регистрирующим моменты времени, когда Perl вошел или вышел из каждой подпрограммы.

Когда профилируемый сценарий завершается, `DProf` сохраняет дамп с информацией о хронометраже в файле с именем *tmon.out*. Программа *dprofpp*, поставляемая с Perl, умеет анализировать *tmon.out* и создавать отчет. Программу *dprofpp* можно также использовать как внешний интерфейс для процесса в целом с помощью ключа *-p*, описываемого далее.

Пусть имеется такая программа:

```
outer();

sub outer {
    for (my $i=0; $i < 100; $i++) { inner() }
}

sub inner {
    my $total = 0;
    for (my $i=0; $i < 1000; $i++) { $total += $i }
}

inner();
```

dprofpp может вывести следующую информацию:

```
Total Elapsed Time = 0.537654 Seconds
  User+System Time = 0.317552 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c Name
 85.0   0.270  0.269    101   0.0027 0.0027 main::inner
 2.83   0.009  0.279      1   0.0094 0.2788 main::outer
```

[1] По крайней мере, так считает Натан Торкингтон, который написал этот раздел книги.

Обратите внимание, что сумма процентов не равна 100. Более того, в нашем случае она довольно далека от этого значения, и это должно послужить подсказкой, что программу следует погонять подольше. Общее правило заключается в следующем: чем больше данных профилирования можно собрать, тем лучше статистическая выборка. Если увеличить число проходов внешнего цикла со 100 до 1000, результаты получатся более точными:

```
Total Elapsed Time = 2.875946 Seconds
  User+System Time = 2.855946 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c Name
 99.3   2.838  2.834   1001   0.0028 0.0028 main::inner
 0.14   0.004  2.828      1   0.0040 2.8280 main::outer
```

В первой строке сообщается, сколько времени выполнялась программа от начала и до конца. Во второй строке выводятся два числа: время, потраченное на выполнение пользовательского кода («user»), и время, потраченное операционной системой на выполнение системных вызовов, осуществляемых из вашей программы («system»). (Не будем обращать внимания на ложную высокую точность представления этих чисел: генератор тактовых импульсов компьютера наверняка не отмечает миллионные доли секунды, в лучшем случае – сотые.)

Время «user+system» можно изменить с помощью параметров командной строки *dprofpp*. Ключ *-r* осуществляет вывод истекшего времени, *-s* – только системного и *-u* – только времени пользователя.

Остальная часть отчета содержит распределение времени по подпрограммам. Строка «Exclusive Times» означает, что, когда подпрограмма `outer` вызывала подпрограмму `inner`, время, затраченное в `inner`, не включалось во время, потраченное в `outer`. Чтобы изменить этот режим и суммировать показатели `inner` и `outer`, передайте *dprofpp* ключ *-I*.

Для каждой подпрограммы сообщаются следующие значения: `%Time` – время в процентах, потраченное внутри этой подпрограммы; `ExclSec` – время в секундах, потраченное в этой подпрограмме, исключая время работы вызванных из нее подпрограмм; `CumulS` – время в секундах, потраченное в этой подпрограмме и вызванных из нее подпрограммах; `#Calls` – число вызовов подпрограммы; `sec/call` – среднее время в секундах каждого вызова подпрограммы, не включая время работы подпрограмм, вызванных из нее; `Csec/c` – среднее время в секундах каждого вызова подпрограммы вместе с работой подпрограмм, вызванных из нее.

Наиболее полезной из этих характеристик является `%Time`, которая показывает, куда уходит время. В нашем случае большую часть времени занимает процедура `inner`, поэтому следует попытаться оптимизировать ее или придумать алгоритм, при котором она вызывалась бы реже. :-) Ключи *dprofpp* предоставляют доступ к другим данным или изменяют способ подсчета времени. Можно также заставить *dprofpp* сначала запустить сценарий, поэтому помнить ключ *-d:Dprof* не обязательно:

-p *SCRIPT*

Сообщает *dprofpp* о необходимости профилировать указанный сценарий *SCRIPT*, а затем интерпретировать данные профилирования. См. также *-Q*.

-Q Используется вместе с *-p*, чтобы заставить *dprofpp* завершить работу после выполнения профилирования без интерпретации данных.

-a Сортировать выводимую информацию в алфавитном порядке по именам подпрограмм, а не по убыванию доли времени.

-R Вести отдельный учет по анонимным процедурам, определенным в том же пакете. По умолчанию все анонимные процедуры учитываются как одна с именем `main::__ANON__`.

-I Вывести время работы всех подпрограмм с учетом времени работы вложенных вызовов подпрограмм.

-l Сортировать по количеству обращений к подпрограмме. Может помочь определить кандидатов на подстановку (inlining).

-O *COUNT*

Показать только первые *COUNT* подпрограмм. По умолчанию равно 15.

-q Не выводить заголовки колонок.

-T Вывести на стандартное устройство вывода дерево вызова подпрограмм. Статистические данные по подпрограммам не выводятся.

-t Вывести на стандартное устройство вывода дерево вызова подпрограмм. Статистические данные по подпрограммам не выводятся. Функция, вызванная несколько раз (последовательно) на одном уровне вызова, выводится один раз с указанием счетчика обращений.

-S Структурировать вывод согласно порядку вызова подпрограмм:

```
main::inner x 1        0.008s
main::outer x 1        0.467s = (0.000 + 0.468)s
  main::inner x 100    0.468s
```

Это нужно понимать так: с верхнего уровня программы `inner` была вызвана один раз и работала 0,008 с, с верхнего уровня программы `outer` была вызвана один раз и работала 0,467 с, с учетом времени, затраченного во вложенных вызовах подпрограмм (0 с в самой `outer` и 0,468 с в подпрограммах, вызванных из `outer`), выполнив `inner` (которая работала 0,468 с) 100 раз. Ну как, все понятно?

Ветви на одном и том же уровне (например, один раз вызвана `inner` и один раз вызвана `outer`) сортируются по суммарному времени.

-U Не сортировать. Выводить согласно порядку обнаружения в исходных данных профилирования.

-v Сортировать по среднему времени нахождения в подпрограмме для всех вызовов. Иногда помогает выявить кандидатов на оптимизацию вручную путем подстановки (inlining) тел подпрограмм.

-g *subroutine*

Игнорировать подпрограммы, исключая *subroutine* и вызовы из нее.

Прочие параметры описаны в *dprofpp*(1), стандартной странице руководства по этой программе.

Devel::NYTProf

Разработку модуля `Devel::NYTProf` начал Адам Каплан (Adam Kaplan) в *New York Times*, но в настоящее время разработка этого модуля ведется за стенами Times.

Это быстрый (написан на C) и мощный профилировщик, способный выводить прекрасно отформатированные отчеты. И это самый быстрый профилировщик инструкций и подпрограмм из доступных на сегодняшний день, однако в книге не так много места, чтобы можно было рассказать обо всех его замечательных возможностях. Загрузите его из архива CPAN и попробуйте использовать, как любой другой отладчик:

```
% perl -d:NYTProf your_program
```

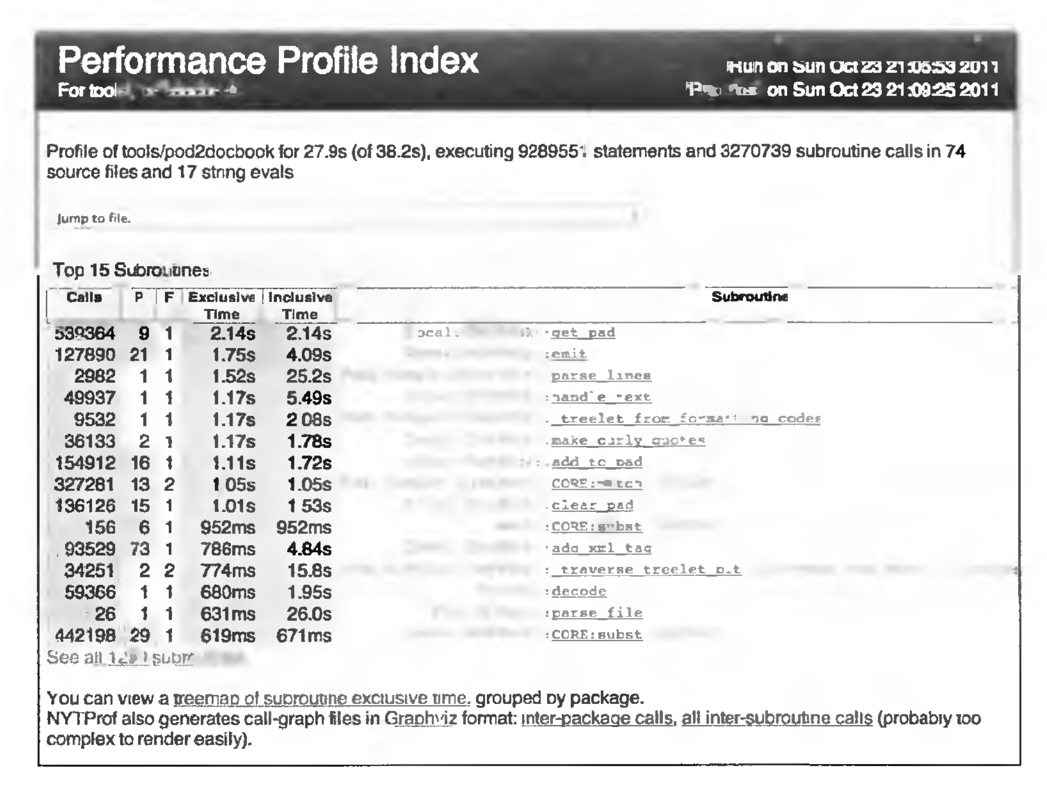
После завершения можно исследовать полученные результаты, которые сохраняются в формате HTML. Первый HTML-файл (рис. 18.1) содержит сводную информацию:

```
% nytprofhtml --open
```

С помощью переменной среды NYTPROF можно определить самые разнообразные настройки профилировщика. Например, можно сообщить ему, когда следует начинать хронометраж: немедленно, на этапе INIT или в начале END:

```
% env NYTPROF=start=init perl -d:NYTProf your_program
```

Дополнительную информацию вы найдете в документации модуля. А теперь сделаем перерыв на чашечку кофе. Это совершенно необходимо, прежде чем приступать к следующей главе.



Calls	P	F	Exclusive Time	Inclusive Time	Subroutine
539364	9	1	2.14s	2.14s	get_pad
127890	21	1	1.75s	4.09s	emit
2982	1	1	1.52s	25.2s	parse_lines
49937	1	1	1.17s	5.49s	handle_text
9532	1	1	1.17s	2.08s	_treelet_from_formatting_codes
36133	2	1	1.17s	1.78s	make_curly_quotes
154912	16	1	1.11s	1.72s	add_to_pad
327281	13	2	1.05s	1.05s	CORE:match
136126	15	1	1.01s	1.53s	clear_pad
156	6	1	952ms	952ms	CORE:subst
93529	73	1	786ms	4.84s	add_xml_tag
34251	2	2	774ms	15.8s	_traverse_treelet_bit
59366	1	1	680ms	1.95s	decode
26	1	1	631ms	26.0s	parse_file
442198	29	1	619ms	671ms	CORE:subst

Рис. 18.1. Начальная страница в формате HTML с результатами работы NYTProf

19

CPAN

Изначально CPAN служил репозиторием программного обеспечения на языке Perl, но с тех пор превратился в коллекцию разноплановых служб, построенных на базе репозитория. Когда люди говорят «CPAN», они могут подразумевать любую из этих служб, так как люди склонны соотносить с этим названием все, что связано с центральным репозиторием.

История

Где-то в конце 1993 года Тим Банс (Tim Bunce), Яркко Хиетаниеми (Jarkko Hietaniemi) и Андреас Кениг (Andreas König) создали список рассылки perl-packrats для обсуждения идеи создания единого архива программного обеспечения, написанного на Perl 4 и разбросанного по Интернету. В этом же году была запущена разработка версии Perl 5, одной из основных особенностей которой должна была стать расширяемая система модулей, позволяющая развивать возможности языка, не изменяя *perl*. Джаред Райн (Jared Rhine) предложил идею централизованного репозитория, но она не нашла отклика. Его идея была основана на опыте CTAN (*http://www.ctan.org*), Comprehensive TeX Archive Network (архив TeX).

Пару лет спустя Яркко вернулся к этой идее и создал FTP-архив с адресом *ftp://ftp.cpan.org*. Вскоре после этого Андреас Кениг (Andreas König) создал PAUSE (*http://pause.perl.org*), Perl Authors Upload Server, дающий людям возможность передавать свои наработки в этот репозиторий. То, что многие подразумевают под названием «CPAN», в действительности является всего лишь двумя каталогами, которые CPAN зеркалирует с сервера PAUSE. Однако CPAN – это намного более обширная сущность.

Службы главного сервера CPAN зеркалируются множеством серверов для обеспечения быстрого доступа из любой точки мира. В настоящее время насчитывается порядка 300 общедоступных зеркал, размещенных на всех шести континентах.

Любой желающий может создать новое общедоступное или даже частное, для собственных нужд, зеркало CPAN.[1]

По мере роста популярности CPAN архив стал обрастать другими проектами. Грэм Барр (Graham Barr) добавил интерфейс поиска *http://search.cpan.org*. Барби (Barbie) воплотил идею CPAN Testers для организации тестирования каждого дистрибутива в CPAN. Дэвид Кантрелл (David Cantrell) разработал CPANdeps для объединения результатов тестирования дистрибутивов со всеми их зависимостями. Мориц Онкен (Moritz Onken) создал поисковый сайт второго поколения на основе проекта MetaCPAN (*https://www.metacpan.org*), являющегося частью проекта Google Summer of Code.

Вокруг собственно архива CPAN, являющегося всего лишь центральным репозиторием, сосредоточено множество других служб.

Обзор репозитория

Большинство файлов, имеющихся в CPAN, попали туда с сервера PAUSE, содержащего каталоги *authors* и *modules*. Помимо каталога с модулями, в CPAN хранится масса другой информации, и ниже вашему вниманию предлагается экскурс по наиболее интересным разделам архива.

authors (http://www.cpan.org/authors)

Этот каталог является зеркалом PAUSE и содержит огромное количество подкаталогов, упорядоченных по идентификаторам авторов, хранящимся в подкаталоге *id* (*http://www.cpan.org/authors/id*). На верхнем уровне в качестве имен каталогов выступают начальные буквы идентификаторов авторов, на втором уровне – пары начальных букв, и на последнем уровне – полные идентификаторы. Например, для автора NANIS (Sinan Ünür) путь к его каталогу имеет вид *id/N/NA/NANIS*. В этом каталоге находится все, что выгрузил и не удалил Sinan (удаленные файлы можно найти на сайте BackPAN: *http://backpan.perl.org*).

У некоторых авторов имена каталогов соответствуют полным именам авторов; например, *Hugo_van_der_Sanden*. Ранее, когда авторов было не так много, каталог *authors* имел более плоскую структуру. С ростом популярности CPAN количество зарегистрированных авторов превысило 9000, и теперь каталог PAUSE разбивает имена авторов в трехуровневую структуру.

doc

В этом каталоге хранится документация по Perl, а также масса различных комментариев к этой документации, но он более не поддерживается официально. В нем можно найти довольно интересные материалы, но в настоящее время он не является основным источником информации о языке Perl. Основную электронную документацию по Perl см. на сайте *http://perldoc.perl.org*, а дополнительную документацию с описанием модулей – на сайтах модулей.

modules

Любопытно, но модули хранятся не здесь. Данный каталог содержит специальные индексные файлы, которые позволяют клиентам CPAN преобразовывать

---

1 См. статью «How to mirror CPAN» (как создать зеркало CPAN) по адресу *http://www.cpan.org/misc/how-to-mirror.html*.

имя пакета, такое как Mojolicious, в путь к файлу в каталоге *authors*, в данном случае – к файлу *authors/id/S/SR/SRI/Mojolicious-1.99.tar.gz* (возможно к другому, в зависимости от номера последней версии Mojolicious).

Здесь также существуют подкаталоги, такие как *by-module*, обеспечивающие организацию по именам модулей, а не по идентификаторам авторов. Элементы в этом каталоге являются символическими ссылками, уходящими вглубь каталога *authors*, где находятся фактические файлы.

Кроме того, имеется каталог *by-category*, не особенно полезный в наши дни. До появления в CPAN многих тысяч дистрибутивов архивариусы CPAN стремились отнести каждый модуль к той или иной категории, благодаря чему имелась возможность навигации по категориям[1] в поисках требуемого модуля. Прямой поиск оказался более популярным, чем блуждание по виртуальным стеллажам CPAN, поэтому раздел «Module List» со списком модулей очень быстро потерял свой глянец и пришел в упадок. Задумка была отчасти в том, чтобы авторы сами регистрировали дистрибутивы и относили их к уместным категориям в PAUSE, но мало кто теперь так поступает.

ports

Этот каталог содержит исходный код и в некоторых случаях готовые исполняемые образы Perl для платформ, которые не поддерживаются непосредственно стандартным дистрибутивом или для которых известную трудность составляет получение компилятора. Эти портированные версии языка являются собственными проектами авторов и не всегда работают точно так, как описано в этой книге. В настоящее время мало какой системе требуется особая сборка Perl. Указатель в этом каталоге в любом случае представляет интерес, поскольку содержит сведения о том, с какого момента времени тот или иной производитель операционных систем начал включать Perl в поставку своей ОС.

scripts

Этот каталог содержит небольшую коллекцию разнообразных программ на Perl. В значительной степени он является пережитком времен, когда люди распространяли уже готовые программы. Авторы включали свои программы в раздел *scripts*, добавляя в них специальные заголовки с описанием. Увы, в настоящее время этого почти никто не делает. Теперь авторы предпочитают выгружать готовые программы в составе своих модулей для Perl, обычно в пространстве имен App::. Архив CPAN не очень хорошо подходит для распространения сценариев, он в большей степени ориентирован на модули.

src

В этом каталоге вы найдете исходный код для стандартного дистрибутива Perl. Точнее, для двух стандартных дистрибутивов Perl. Один из них помечен как *maint* (поддерживаемый), а другой как *devel* (разрабатываемый). Обычно разработка Perl делится на две ветви. Первая, *maint*, предназначена для использования в промышленном окружении. Ветвь *devel* – экспериментальная, в ней разработчики опробуют новые особенности, новый код и многое другое, что пока не отличается стабильностью в работе.

Чтобы отличить одно от другого, взгляните на номер версии, такой как 5.14.2. Первое число – это старший номер версии, подразумевает Perl v5. Второе чис-

[1] Помнит ли кто-нибудь, как выглядел Yahoo! до взрыва популярности поиска?

ло – младший номер версии.[1] Если младший номер является четным числом – это поддерживаемая версия. То есть, дистрибутивы с номерами версий 5.10.1, 5.12.4 и 5.14.2 являются стабильными версиями, поскольку 10, 12 и 14 – четные числа. Если младший номер версии является нечетным числом, как в версии 5.15.3, – это экспериментальная версия, поскольку 15 – число нечетное.

В этом каталоге всегда есть две ссылки на самые последние версии обеих веток. Ссылки *latest.tar.gz* (*http://www.cpan.org/src/latest.tar.gz*) и *maint.tar.gz* (*http://www.cpan.org/src/maint.tar.gz*) указывают на самую последнюю версию и на самую свежую поддерживаемую версию соответственно[2]. Специалисты, осуществляющие сопровождение CPAN, не приветствуют использование этой терминологии, потому что многие не понимают заложенного в нее смысла.

Создание собственного мини-зеркала CPAN

Любой желающий может создать собственное зеркало CPAN, но вы должны понимать, что при этом вам придется синхронизировать 24000 дистрибутивов, занимающих порядка 13 Гбайт дискового пространства (на момент написания этих строк). Поскольку репозиторий PAUSE индексирует только последние дистрибутивы, большая часть архива вам, вероятно, не потребуется. В большинстве случаев достаточно хранить только самые последние версии. Учитывая все это, в 2002 году Рэндал Шварц (Randal Schwartz) создал программу *minicpan* и написал об этом в журнале «Linux Magazine».[3] Ему удалось уменьшить объем своего локального зеркала CPAN на 80%, из-за чего брайан д фой (brian d foy) ввел термин «коэффициент Шварца», определяющий это отношение.

Все необходимые инструменты содержатся в дистрибутиве `CPAN::Mini`. Этот модуль не является частью стандартной библиотеки, поэтому его придется установить вручную (процесс описан далее в этой главе).

Сначала необходимо выполнить настройки, указав, откуда будут извлекаться новые данные и где они будут сохраняться:

```
local: /Users/Amelia/MINICPAN
remote: http://cpan.example.com/
```

Запустите программу *minicpan*, чтобы создать тонкий срез репозитория:

```
% minicpan
Using config from /Users/Amelia/.minicpanrc
Updating /Users/Amelia/MINICPAN
Mirroring from http://cpan.example.com/
==============================================================
authors/01mailrc.txt.gz ... updated
modules/02packages.details.txt.gz ... updated
modules/03modlist.data.gz ... updated
```

---

[1] *Младший* – не значит «незначительный». Изменение этого числа отмечает выпуски значительных версий. Вообще говоря, первое число в номере версии следует интерпретировать как «для Perl5», а следующее число – как старший номер версии.

[2] Официально разработчики Perl поддерживают две последние версии. Если текущей версией является Perl v5.16, это означает, что версия v5.14 также официально поддерживается, а версия v5.12 – нет. См. подробности на страницах *perlpolicy*.

[3] *http://www.stonehenge.com/merlyn/LinuxMag/col42.html.*

```
authors/id/A/AA/AAR/Math-Clipper-1.01.tar.gz ... updated
authors/id/A/AA/AAR/CHECKSUMS ... updated
```

Настройте клиентов CPAN на использование этого репозитория, и после этого вы получите возможность устанавливать модули, даже находясь в поезде, самолете или автомобиле, посреди пустыни Блэк-Рок (Black Rock Desert) и даже там, где нет электричества. Ну... пока остается заряд в аккумуляторах.

Вы можете с большой степенью точности управлять содержимым своего репозитория, однако для этого придется написать собственную программу *minicpan*, реализующую необходимое управление. Функции `module_filters` и `path_filters` позволяют использовать регулярные выражения или ссылки на подпрограммы, определяющие, какие модули или каких авторов следует пропустить. Если попытка сопоставления с шаблоном или вызов подпрограммы возвращает истинное значение, модуль `CPAN::Mini` пропускает такой дистрибутив:

```
use CPAN::Mini;

CPAN::Mini->update_mirror(
    remote         => "http://cpan.mirrors.comintern.su",
    local          => "/usr/share/mirrors/cpan",
    force          => 0,
    module_filters => [ qr/Acme/i ],
    path_filters   => [
        qr/RJBS/,
        sub { $_[0] =~ /SUNGO/ }
    ],
);
```

Экосистема CPAN

Архив CPAN в действительности является комплексом служб, стоящих между теми, кто выгружает дистрибутивы, и теми, кто устанавливает их. В этой главе мы не будем детально рассматривать все службы, а просто познакомимся с самыми основными элементами.

PAUSE

PAUSE (*http://pause.perl.org*) – это вход в архив CPAN для авторов. Прежде чем что-либо опубликовать, необходимо зарегистрироваться. Делается это легко и просто. Один из администраторов PAUSE проверит ваш запрос, только чтобы убедиться, что вы не бот, и затем настроит учетную запись.

Получив учетную запись, вы сможете помещать свои файлы в репозиторий PAUSE. Публиковать можно все, что заблагорассудится. PAUSE никак не интересуется тем, что вы делаете или как вы это делаете.

Когда вы публикуете свой дистрибутив, программа индексирования репозитория PAUSE исследует ваш архив на наличие пространств имен Perl. На использование пространств имен нет никаких ограничений, но в PAUSE хранится список лиц, которые уполномочены изменять те или иные пространства имен.

- Первый, кто использовал некоторое пространство имен, получает статус *первооткрывателя* и становится *основным хранителем* (*primary maintainer*).

- *Основной хранитель* может выдавать права *сохранителей* (*co-maintainer*) другим авторам.
- Основной хранитель может передать свой статус другому автору.

Если публикуемый вами дистрибутив использует пространство имен, изменять которое вы не уполномочены, программа индексирования PAUSE откажется индексировать модуль и отправит вам сообщение об ошибке. При этом ваш дистрибутив будет сохранен и останется доступным в CPAN. Люди смогут загружать его. Но, не будучи проиндексирован, дистрибутив не попадет в базу данных PAUSE. Если ваш дистрибутив не появится в базе данных, клиенты CPAN не узнают о его существовании и не смогут установить его. Репозиторий PAUSE просто не заметит ваш дистрибутив, как и весь остальной мир.

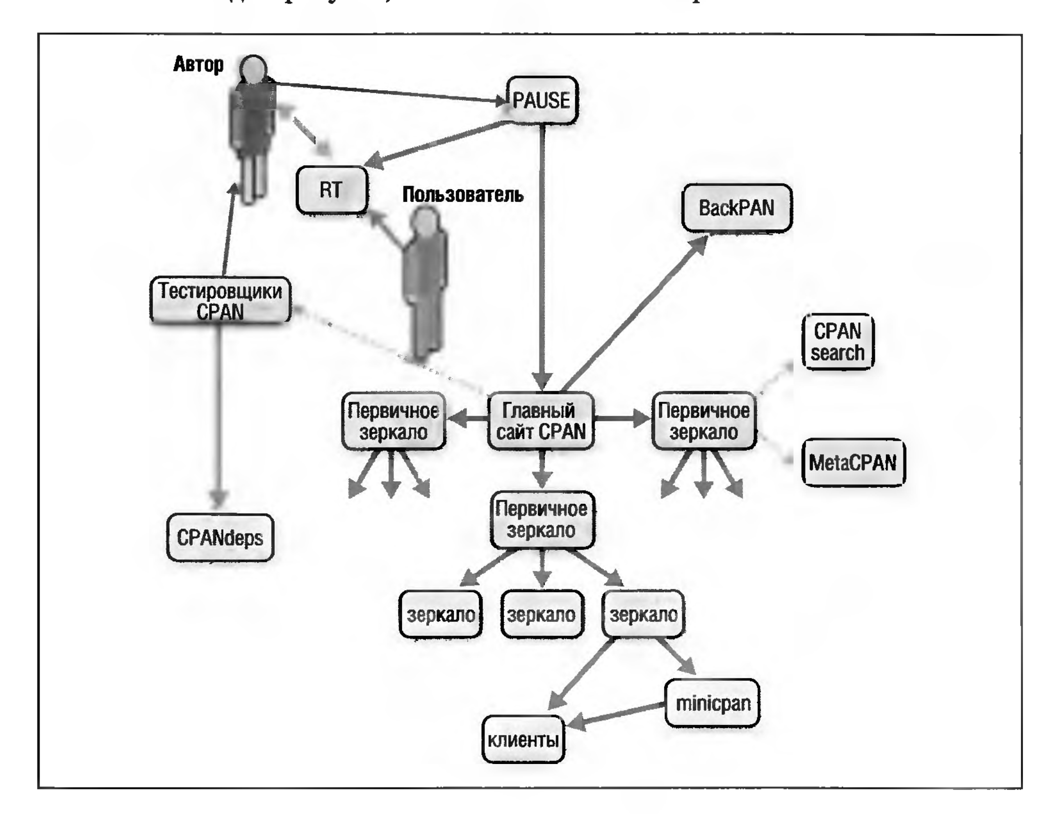


Рис. 19.1. *Карта экосистемы CPAN*

Поиск в CPAN

Существует два основных сайта, осуществляющих поиск в архиве CPAN, и оба предоставляют аналогичные возможности. Каждый дополнительно служит для хранения специфических для каждого дистрибутива ссылок на другие проекты CPAN:

- CPAN Search (*http://search.cpan.org*)
- MetaCPAN (*https://www.metacpan.org*)

Тестирование

В Perl поддерживается высокая культура тестирования. Как только кто-то опубликовал новый дистрибутив в репозитории PAUSE, свободная конфедерация машин разных форм и размеров, известная как CPAN Testers (*http://testers.cpan.org*), загружает, собирает и выполняет прогон тестовых наборов этого дистрибутива. Данная группа старается проверить как можно большее число модулей на как можно большем числе платформ и вариантов Perl.

Благодаря этому автор может вести разработку на одной платформе, а выгрузив свои наработки, получить результаты тестирования на других платформах и с разными версиями Perl. И все это бесплатно! Получить подробные инструкции по созданию дистрибутивов, «дружественных для проекта Testers», можно на вики-странице CPAN Testers (*http://wiki.cpantesters.org*).

Результаты тестирования доступны также пользователям модулей из CPAN. Любой желающий может исследовать отчеты с результатами тестирования и увидеть, как чувствует себя модуль. Проект CPANdeps (*http://deps.cpantesters.org*) Дэвида Кантрелла (David Cantrell) предоставляет отчеты с результатами тестирования в табличной форме с перечислением платформ и версий Perl, а также позволяет получить сводную информацию для всех зависимостей модуля, в качестве меры «вероятности успеха» установки.

Отслеживание ошибок

Поскольку Perl и CPAN не являются единым централизованным проектом, существует множество мест, где можно оставлять отчеты об ошибках или изучать их. Однако многие сообщают об ошибках в частной форме, направляя персональные электронные письма, т. е. не создавая общедоступные записи, над которыми могли бы работать все желающие, комментируя или даже исправляя эти ошибки. Открытое сообщество может решать проблемы, только если имеет открытый доступ к ним, а персональное электронное письмо закрыто для всего остального мира.

rt.cpan.org

Архив CPAN, являющийся репозиторием, где хранятся тысячи проектов различных авторов, тоже имеет свой инструмент слежения за ошибками. Для каждого дистрибутива выделяется отдельная очередь в Request Tracker (*https://rt.cpan.org*). Это первый способ сообщать о проблемах в модуле.

Другие средства слежения за ошибками

Некоторые авторы модулей предпочитают использовать другие механизмы отслеживания ошибок, отличные от *https://rt.cpan.org*. Определить их предпочтения можно, прочитав документацию к модулю. Иногда авторы модулей включают инструкции в документацию к своим модулям, а иногда нет. Поскольку файлы вроде *README* и *META.yml* удаляются во время установки, попробуйте отыскать их на одном из сайтов проекта CPAN Search.

perlbug

Чтобы сообщить об ошибке в модуле, поставляемом вместе с *perl*, можно воспользоваться утилитой *perlbug*. Она соберет информацию о вашей платформе и интер-

претаторе, чтобы те, кто будет заниматься диагностикой ошибки, имели всю необходимую информацию. В действительности эта утилита является интерфейсом для отправки особым образом форматированных электронных писем на адрес *perlbug@perl.org*, куда вы можете отправить письмо собственноручно, если пожелаете. Эти отчеты автоматически направляются в список рассылки Perl 5 Porters. Некоторые модули *живут двойной жизнью*, входя в состав *стандартной библиотеки* и в CPAN, поэтому иногда бывает непросто выбрать правильное место для отправки отчета. Однако это не должно мешать вам отправлять сообщения о проблемах. Отправляйте в *perlbug*, а мы переправим его при необходимости.

rt.perl.org

perlbug отправляет отчеты об ошибках на сайт Request Tracker (*https://rt.perl.org*). Там же вы сможете ознакомиться со списком обнаруженных ошибок, включая ошибки в модулях из стандартной библиотеки. Посетите этот сайт, если вы ничего не обнаружите на сайте *https://rt.cpan.org*.

Установка модулей из CPAN

Существует две основных системы сборки, которые обычно используются авторами при создании дистрибутивов для CPAN. Одна из них основана на утилите *make*, а другая – исключительно на языке Perl.

Вручную

Дистрибутивы из CPAN редко устанавливаются вручную, так как при этом пришлось бы вручную разрешить все зависимости, что достаточно утомительно. Однако это вполне возможно, и знание этой процедуры может пригодиться в будущем.

Заглянув внутрь дистрибутива, вы наверняка обнаружите файл *Makefile.PL* или *Build.PL*. Порядок их использования представлен в табл. 19.1.

Таблица 19.1. Команды сборки при использовании двух основных инструментов

Makefile.PI	Build.PL
`% perl Makefile.PL`	`% perl Build.PL`
`% make`	`% ./Build`
`% make test`	`% ./Build test`
`% make install`	`% ./Build install`

С настройками по умолчанию обе системы попытаются установить дистрибутив в вашу библиотеку, с путями, настроенными вами (или кем-то другим) при установке *perl*. Увидеть эти каталоги можно в конце вывода команды *perl -V*.

У вас может оказаться недостаточно прав для записи в эти каталоги, но вы с таким же успехом можете выполнить установку в любой другой каталог по выбору, сообщив его файлу сборки. Поведение файлов сборки можно изменять посредством параметров командной строки или переменных среды.

```
% perl Makefile.PL INSTALL_BASE=/some/other/directory

% perl Build.PL --install_base /some/other/directory
```

Если определить параметры в переменных среды, их не потребуется указывать повторно при установке очередного дистрибутива. Каждая система сборки имеет свои переменные среды для хранения параметров командной строки, принимаемых по умолчанию. Ниже показано, как настроить эти переменные в оболочке */bin/sh*:

```
% export PERL_MM_OPT='INSTALL_BASE=/some/other/directory'
% export PERL_MB_OPT='--install_base /some/other/directory'
```

А вот как выполнить те же настройки в оболочке *csh*:

```
% setenv PERL_MM_OPT 'INSTALL_BASE=/some/other/directory'
% setenv PERL_MB_OPT '--install_base /some/other/directory'
```

Неважно, каким способом вы сообщите файлу сборки, куда следует выполнить установку. В любом случае, в конец каждого указанного пути будет добавлен подкаталог *lib/perl5*[1]. Запомните это, так как это знание пригодится в следующей части.

Если модули устанавливаются в другой каталог, не забудьте сообщить программам, где следует искать их, либо с помощью ключа *-I*:

```
% perl -I/some/other/directory/lib/perl5 program.pl
```

либо с помощью переменной среды PERL5LIB:

```
% export PERL5LIB=/some/other/directory/lib/perl5
% perl program.pl
```

Для этой цели в программах можно также использовать прагму lib:

```
use lib qw(/some/other/directory/lib/perl5);
```

Если вы не помните эти пути, воспользуйтесь модулем local::lib из CPAN (он не входит в стандартный дистрибутив Perl). Загрузив самого себя, он сообщит, какое значение использовать. По умолчанию он использует подкаталоги в домашнем каталоге:

```
% perl -Mlocal::lib
export PERL_LOCAL_LIB_ROOT="/home/amelia";
export PERL_MB_OPT="--install_base /home/amelia/perl5
export PERL_MM_OPT="INSTALL_BASE=/home/amelia/perl5";
export PERL5LIB="/home/amelia/perl5/lib/perl5/darwin-2level:/home/amelia/perl5/lib/
perl5";
export PATH="/Users/amelia/perl5/bin:$PATH";
```

Можно также указать другой каталог:

```
% perl -Mlocal::lib=/some/other/directory
export PERL_LOCAL_LIB_ROOT="/some/other/directory";
export PERL_MB_OPT="--install_base /some/other/directory";
export PERL_MM_OPT="INSTALL_BASE=/some/other/directory";
export PERL5LIB="/some/other/directory/lib/perl5/darwin-2level:/some/other/
directory/lib/perl5";
export PATH="/some/other/directory/bin:$PATH"
```

Вам все равно придется настроить переменные среды для себя, однако для программ будет вполне достаточно простого использования local::lib:

[1] Это поведение по умолчанию. Его можно изменить, определив настройки *Configure* с помощью ключа -Dinstallstyle при компиляции *perl*.

```
use local::lib;

use local::lib qw(/some/other/directory);
```

Клиенты CPAN

Большинство предпочитают устанавливать модули с помощью программы-клиента.[1] Существует три популярных клиента CPAN, и каждый разрабатывался для определенного круга пользователей со своими особыми потребностями. Необязательно все время использовать один и тот же клиент, и выбор клиента не является пожизненным.

cpan

Команда *cpan*, входящая в состав стандартной библиотеки и модуля CPAN.pm, обеспечивает простой и быстрый способ установки модулей. Просто укажите ей. какой модуль требуется установить:

```
% cpan IO::Interactive AnyEvent
```

Чтобы установить модули в другой каталог, необходимо выполнить соответствующие настройки. Если выполнить команду *cpan* без аргументов, она запустит интерактивную оболочку CPAN.pm:

```
% cpan
cpan> o conf makepl_arg INSTALL_BASE=/some/other/directory
cpan> o conf mbuild_arg "--install_base /some/other/directory"
cpan> o conf commit
```

Запустить оболочку CPAN.pm можно также командой:

```
% perl -MCPAN -e shell
cpan> install POE
```

или воспользоваться модулем local::lib:

```
% perl -MCPAN -Mlocal::lib -e shell
cpan> install Set::CrossProduct
```

cpanp

В состав Perl входит еще один интерфейс к архиву CPAN. CPANPLUS. Целью этого проекта было учесть уроки разработки CPAN.pm, а запускается этот интерфейс так:

```
% cpanp -i IO::Interactive AnyEvent
```

Можно также запустить интерактивную оболочку CPANPLUS:

```
% perl -MCPANPLUS -e shell
CPAN Terminal> install POE
```

Интерфейс CPANPLUS использует систему настройки, управляемую посредством меню, поэтому, запустив ее в интерактивной оболочке, достаточно просто следовать ее подсказкам.

---

[1] Или с помощью диспетчера пакетов, предоставляемого операционной системой.

cpanminus

Третий популярный клиент, который может вам понравиться, если вас удовлетворят его умолчания, – это *cpanminus*, или просто *cpanm*. Это минималистичный клиент, стремящийся соответствовать требованиям большинства пользователей. Он использует модуль `local::lib` по умолчанию. Многие предпочитают именно эту программу, и она отлично подойдет вам, пока не потребуется что-то необычное.

Поскольку отличительной чертой *cpanm* является простота использования, клиент не требует установки дополнительных модулей.[1] Просто загрузите его и приступайте к использованию. Документация *cpanm* (*https://www.metacpan.org/module/App::cpanminus*) демонстрирует, как это сделать с помощью *curl*,[2] с последующей передачей каталога через конвейер команде *perl*, чтобы превратить его в *cpanm*. Это предпочтительный способ, потому что в этом случае *cpanm* сможет извлечь необходимые настройки из *perl*:

```
% curl -L http://cpanmin.us | perl - App::cpanminus
```

Или сохраните его как *cpanm* после загрузки и запустите. В UNIX это (по сути) то же самое, что сохранить результат превращения дистрибутива в выполняемый файл, однако в данном случае будет использоваться команда */usr/bin/env* для поиска первой попавшейся версии *perl*:

```
% cd ~/bin
% curl -LO http://xrl.us/cpanm
% chmod +x cpanm
```

Сохранив *cpanm*, его можно использовать для установки модулей:

```
% cpanm HTML::Barcode
```

Создание дистрибутивов для CPAN

Это короткое введение в создание дистрибутивов для распространения через CPAN. Этой теме можно посвятить целую книгу.[3] Книга «Intermediate Perl»[4], один из учебников по языку Perl, выпущенных издательством O'Reilly, охватывает эту тему намного подробнее.

---

1 Некоторые считают, что стандартная библиотека в действительности представляет собой лишь начальный комплект модулей, и поэтому вам может потребоваться использовать *cpan* или *cpanp*. Посмотрим, как это скажется на будущих версиях. Кстати, это одна из интереснейших тем, способных оживить самую унылую дискуссию пользователей Perl. Просто упомяните об этом и отступите на шаг, чтобы понаблюдать за жаркими баталиями.

2 *curl* – инструмент командной строки для передачи данных (*http://curl.haxx.se*).

3 И такая книга была написана: Sam Tregar «Writing Perl Modules for CPAN» (*http://www.apress.com/9781590590188*), Apress.

4 Рэндал Л. Шварц, брайан д фой, Том Феникс «Perl: изучаем глубже», 3-е издание». – Пер. с англ. – СПб.: Символ-Плюс, 2014.

Приступая к созданию дистрибутива

Архив CPAN существует достаточно давно. С тех пор появилось множество устоявшихся приемов и соглашений, поэтому на сегодняшний день практически все согласны с перечнем того, что должно входить в дистрибутив. Вам не придется начинать с чистого листа, если вы воспользуетесь инструментами для создания заготовки своего дистрибутива.

h2xs

Канонический инструмент создания дистрибутивов, который в действительности не является инструментом создания дистрибутивов. Как следует из названия, он предназначен для преобразования заголовочных файлов на языке C в файлы на языке XS, промежуточном языке, связывающем Perl и C. Он продолжал развиваться долгие годы, даже при том что большинством применялся не по основному назначению:

```
% h2xs -XAn Some::Module
Defaulting to backward compatibility with perl 5.14.2

Writing Some-Module/lib/Some/Module.pm
Writing Some-Module/Makefile.PL
Writing Some-Module/README
Writing Some-Module/t/Some-Module.t
Writing Some-Module/Changes
Writing Some-Module/MANIFEST
```

Distribution::Cooker

Модуль `Distribution::Cooker` принадлежит к числу простейших инструментов создания дистрибутивов и проектировался для тех, чьи требования не слишком высоки.[1] Он подготавливает каталог с шаблонами, позволяя вам проектировать ваш дистрибутив по своему усмотрению и затем тиражировать свои настройки. Когда все будет готово, вам не потребуется каждый раз изменять вывод других инструментов. И верно, оптимальный способ использовать `Distribution::Cooker` – начать с изменения вывода других инструментов, пока не будет получен желаемый результат, а затем переходить к проектированию соответствующего шаблона.

Module::Starter

Модуль `Module::Starter` – лучший инструмент для тех, кто пока не знает, чего хочет. Вы начинаете с создания файла настройки в *${HOME}/.module-starter/config* и совсем немного набираете на клавиатуре:

```
author: Amelia Camel
email: amelia@example.com
builder: Module::Build
verbose. 1
```

Затем запускаете *module-starter* и получаете базовую структуру дистрибутива:

---

[1] Модуль `Distribution::Cooker` попал в эту книгу только потому, что написан и используется одним из авторов книги.

```
% module-starter --module=Some::Module2
Created Some-Module
Created Some-Module/lib/Some
Created Some-Module/lib/Some/Module2.pm
Created Some-Module/t
Created Some-Module/t/pod-coverage.t
Created Some-Module/t/pod.t
Created Some-Module/t/manifest.t
Created Some-Module/t/boilerplate.t
Created Some-Module/t/00-load.t
Created Some-Module/ignore.txt
Created Some-Module/Build.PL
Created Some-Module/Changes
Created Some-Module/README
Created Some-Module/MANIFEST
Created starter directories and files
```

Обратите внимание на тестовый файл *Some-Module/t/boilerplate.t*. Это то место, куда можно обратиться, чтобы увидеть изменившиеся значения по умолчанию, такие как описание модуля.

Dist::Zilla

Модуль `Dist::Zilla` – достаточно сложный инструмент, который умеет гораздо больше, чем просто создавать заготовку дистрибутива. Он позволяет управлять полным жизненным циклом модуля, от идеи до воплощения, а затем выпуском, тестированием, исправлением ошибок и повторным выпуском. Он слишком сложный, чтобы описывать его здесь, но многим он нравится.

Тестирование модулей

Высокая культура тестирования – одна из самых привлекательных особенностей языка Perl. Мы уже упоминали о CPAN Testers, людях, которые тестируют все дистрибутивы, имеющиеся в CPAN, на различных платформах. Вам, как автору, самому решать, как реализовать свои тесты. Мы не собираемся рассказывать здесь обо всем, что имеет отношение к тестированию, потому что об этом немало рассказывается в других книгах, таких как «Intermediate Perl» и «Perl Testing: A Developer's Notebook».

Внутреннее тестирование

Если вы пользуетесь одним из двух стандартных инструментов сборки дистрибутивов, инструменты тестирования у вас уже имеются. Запустить тестирование можно командами:

```
% make test
```

```
% ./Build test
```

Обе они работают одинаково: просматривают содержимое файла *test.pl* или каталога *t/*. Файл *test.pl* – это пережиток прошлого, когда все тесты хранились в единственном файле. Подкаталог *t/* – это более удобный подход, потому что в нем можно сохранить множество тестовых файлов с расширением *.t*, которые будут выполнены механизмом тестирования.

Каждый файл с тестами – это обычная программа на языке Perl, обычно использующая в своей работе модуль Test::More. Ниже приводится пример тестового файла, который загружает модуль Math::MySum и тестирует его метод my_sum:

```
use strict;
use warnings;
use Test::More;

BEGIN { use_ok( 'Math::MySum' ) }
can_ok( "Math::MySum", "my_sum" );

my($i, $j) = (1, 3);
my $string = "Amelia";

is($i + $j, Math::MySum->my_sum( $i, $j )
    "Sum of $i and $j is 4");

like($string, qr/mel/, "String has mel in it");

done_testing;
```

Эти программы выводят результаты тестирования в формате TAP (Test Anywhere Protocol – универсальный протокол тестирования), простом текстовом формате, разработанном Ларри и дополненном другими.[1] Ниже представлен вывод этой программы в формате TAP:

```
ok 1 - use Math::MySum;
ok 2 - Math::MySum->can('my_sum')
ok 3 - Sum of 1 and 3 is 4
ok 4 - String has mel in it
1..4
```

Имеется также возможность запускать тесты по отдельности, используя модуль blib для автоматического добавления библиотек сборки в @INC:

```
% perl -Mblib t/failingtest.t
```

Кроме того, можно задействовать инструмент *prove*:

```
% prove -vb t/failingtest.t
```

Самое важное при создании комплектов тестов – охватить тестированием весь код. Поскольку вы являетесь автором и модуля, и тестов, вы легко можете обеспечить прохождение тестов, исключив тестирование наиболее сложных частей модуля. Чтобы убедиться, что это не так, воспользуйтесь модулем Devel::Cover и его программой *cover*, измеряющей процент покрытия программного кода тестами:

```
% cover -test
```

Команда *cover* автоматически выполнит комплект тестов, соберет статистическую информацию и выведет отчет:

---

[1] Многие другие языки программирования заимствовали протокол TAP. Программа, производящая вывод в формате TAP, необязательно должна быть написана на том же языке, что и программа, анализирующая результаты в формате TAP.

```
Reading database from ./cover_db

---------------------------- ------ ------ ------ ------ ------ ------
File                           stmt   bran   cond    sub   time  total
---------------------------- ------ ------ ------ ------ ------ ------
blib/lib/Some/Module.pm        82.9   50.0   27.3   92.3   83.0   72.7
---------------------------- ------ ------ ------ ------ ------ ------

Writing HTML output to ./cover_db/coverage.html .
done.
```

Она измеряет покрытие четырех видов:

инструкции

Выполнение каждой инструкции.

ветви

Тестирование каждой ветви, например, когда инструкция if имеет несколько блоков elsif, каждый из которых рассматривается как отдельная ветвь.

условия

Тестирование каждой комбинации условий, если возможно множество комбинаций. Например, ниже демонстрируется такое условие в операторе if:

```
if ($m && $n) { ... }
```

Этот оператор if имеет три тестируемых комбинации: когда $m и $n имеют истинные значения; когда $m может иметь ложное значение – и в этом случае значение $n не принимается во внимание; когда $m может иметь истинное значение, а $n – истинное или ложное. Тестирование должно охватывать каждую из этих ситуаций.

подпрограммы

Вызов каждой подпрограммы, что также является частью показателя, учитывающего выполнение каждой инструкции.

Внешнее тестирование

Когда дистрибутив будет опубликован в репозитории PAUSE, его автоматически «подхватит» и протестирует инфраструктура CPAN Testers, после чего вам будут высланы результаты тестирования. Это удобно для тестирования на платформах и версиях Perl, которых у вас нет. Для этого вам не придется делать ничего особенного.

Однако такая услуга доступна только для свободно распространяемых дистрибутивов. Если вы не планируете передавать окончательную версию в CPAN, вы все равно можете обеспечить внешнее тестирование, настроив собственную систему CPAN Testers на своей ферме тестовых машин. Для этого можно использовать те же инструменты, которыми пользуется система CPAN Testers, но при этом они будут загружать дистрибутивы из ваших частных источников.

Вы можете также интегрировать тестирование программ на Perl в одну из систем непрерывной интеграции, такую как *smolder* (созданную специально для Perl, но не ограниченную им), Hudson, Jenkins или TeamCity. Любой инструмент, понимающий отчеты в формате TAP (их существует множество), может использоваться для анализа вывода ваших тестов.

IV

Perl как культура

20

Защита данных

Всякий раз, имея дело с пользователем, набирающим команды на клавиатуре, или получая от кого-то информацию по сети, следует осторожно обращаться с данными, поступающими в вашу программу, поскольку другой человек может, по злому умыслу или случайно, послать вам нечто такое, что причинит вред. Perl предоставляет специальный механизм проверки безопасности, носящий название *режим меченых данных* (*taint mode*), назначением которого является изоляция данных, поступивших в вашу программу извне, чтобы вы не смогли сделать с их помощью что-то такое, чего делать не собирались. Например, доверившись по ошибке содержащей имя файла строке сомнительного происхождения, можно добавить запись в файл паролей, думая, что это файл журнала. Подробнее механизм режима меченых данных описывается в разделе «Обработка ненадежных данных».

В многозадачной среде закулисные действия невидимых актеров могут повлиять на безопасность вашей собственной программы. Тот, кто полагает, что единолично владеет внешними объектами (в особенности файлами), как если бы его процесс был единственным в системе, подвергает себя значительно менее очевидным опасностям, чем те, что возникают вследствие работы с данными или кодом сомнительного происхождения. Perl оказывает кое-какую помощь, выявляя ряд ситуаций, которые мы не можем контролировать; что касается ситуаций, которые мы контролировать можем, тут главное – понимать, какие подходы надежно противодействуют невидимому вмешательству. Эти вопросы обсуждаются в разделе «Обработка ошибок синхронизации».

Если извне мы получаем не данные, а фрагмент кода для выполнения, следует проявлять еще большую осторожность, чем с данными. Perl поддерживает некоторые проверки, позволяющие перехватить скрытый код, маскирующийся под данные, чтобы ненароком не выполнить его. Если вы все же хотите выполнить посторонний код, модуль `Safe` позволит поместить подозрительный код в карантин, где он не причинит никакого вреда и, может быть, сделает что-нибудь полезное. Это темы раздела «Обработка ненадежного кода».

Обработка ненадежных данных

Perl облегчает безопасное программирование, даже когда вашу программу использует кто-то, кому можно доверять еще меньше, чем самой программе. Это значит, что некоторые программы должны ограничивать права своих пользователей. В UNIX в эту категорию попадают программы setuid и setgid, как и программы, работающие в привилегированных режимах в других операционных системах, поддерживающих такие понятия. Даже в системах, которые их не поддерживают, те же принципы применимы к сетевым серверам и любым программам, выполняемым этими серверами (например, сценариям CGI, процессорам почтовых списков рассылки и демонам, перечисленным в */etc/inetd.conf*). Все такие программы требуют более пристального внимания, чем обычные.

Даже программы, выполняемые из командной строки, иногда являются хорошими кандидатами на включение режима проверки меченых данных, особенно если их будет выполнять привилегированный пользователь. Программы, обрабатывающие сомнительные данные, например собирающие статистику по файлам журналов или получающие удаленные данные с помощью `LWP::*` и `Net::*`, следует запускать с явным включением режима проверки меченых данных; программы, не проявляющие осторожность, рискуют превратиться в «троянского коня». Поскольку, идя на риск, программы не получают порции адреналина, у них нет особых оснований не быть осторожными.

В сравнении с интерпретаторами командной строки UNIX, которые, в сущности, лишь создают условия для вызова других программ, на Perl легко писать защищенные программы, поскольку он прост и самодостаточен. В отличие от большинства языков программирования командных интерпретаторов, основанных на многократной обработке каждой строки сценария с таинственными подстановками, Perl использует более общепринятую схему вычислений с меньшим количеством тайных подвохов. Кроме того, поскольку в Perl больше встроенных функций, он меньше полагается на внешние (и, возможно, не заслуживающие доверия) программы для выполнения своих задач.

В UNIX, откуда Perl родом, предпочтительный способ компрометации системы безопасности – обманом заставить привилегированную программу сделать что-то, для чего она не предназначалась. Чтобы предотвратить такие атаки, Perl приобрел уникальный метод противостояния враждебному окружению. Perl автоматически включает режим проверки меченых данных, когда обнаруживает, что текущий (effective) идентификатор пользователя или группы программы отличается от реального (real).[1] Даже если для файла сценария Perl не установлены биты setuid и setgid, этот сценарий все равно может быть переведен в режим меченых данных. Это произойдет, если сценарий будет вызван другой программой, которая *сама* выполнялась с другим идентификатором. Программы на Perl, не предназначавшиеся для работы в режиме проверки меченых данных, могут досрочно прекращать свое существование, будучи застигнутыми в момент нарушения правил

---

[1] Бит setuid в правах доступа UNIX имеет значение 04000, а бит setgid имеет значение 02000; можно установить один из них или оба, чтобы дать пользователю программы некоторые права владельца (или владельцев) программы. (В совокупности их называют программами set-id.) Другие операционные системы могут давать программам особые права другими способами, но принцип остается тем же.

безопасности. И это правильно, ведь именно такие интриги исторически плелись в сценариях интерпретатора команд для компрометации защиты системы. Perl не столь доверчив.

Режим проверки меченых данных можно активировать явно, с помощью ключа командной строки -T. Это нужно делать для демонов, серверов и любых программ, работающих от чужого имени, например сценариев CGI. Программы, которые можно запускать удаленно и анонимно с любой машины в Сети, выполняются в самом враждебном из возможных окружений. Не бойтесь сказать «Нет!». Вопреки распространенному мнению, можно проявлять значительную осторожность и не стать при этом усохшим морщинистым ханжой.

Для сайтов, заботящихся о безопасности, выполнение всех сценариев CGI с флагом -T не просто хороший тон: это закон. Мы не утверждаем, что запуск в режиме проверки меченых данных сделает сценарий безопасным, поскольку это не так. Простое перечисление того, что сделает сценарий безопасным, займет целую книгу. Но если вы не используете режим проверки меченых данных при выполнении своих сценариев CGI, значит, без всякой необходимости пренебрегли самой сильной защитой, которую может дать Perl.

В этом режиме Perl предпринимает особые меры предосторожности, называемые *проверками меченых данных* (*taint checks*), чтобы избежать очевидных и скрытых ловушек. Некоторые из этих проверок довольно просты, например отсутствие опасных переменных среды, запрет на запись посторонними в каталоги, перечисленные в переменной среды PATH; но осторожные программисты всегда использовали такие проверки. Есть, однако, другие проверки, которые лучше всего поддерживаются самим языком, именно они и делают привилегированную программу на Perl более надежной, чем соответствующая программа на C, а сценарий CGI на Perl более надежным, чем написанный на языке без проверки меченых данных. (А таковым, насколько нам известно, является любой язык, кроме Perl.)

Принцип прост: нельзя использовать данные, полученные извне выполняемой программы, для воздействия на что-либо еще, находящееся за ее пределами, во всяком случае, намеренно. Любые данные, поступающие в программу извне, «помечаются» как данные сомнительного происхождения (tainted), в том числе все аргументы командной строки, переменные среды и данные, прочитанные из файлов. Меченые данные нельзя использовать прямо или косвенно в операциях, порождающих вложенный интерпретатор команд, а также в операциях, модифицирующих файлы, каталоги или процессы. Любая переменная, устанавливаемая в выражении с участием меченого значения, сама становится меченой, даже если логически невозможно, чтобы меченое значение повлияло на переменную. Однако использование меченой переменной для выбора немеченого значения не приводит к отметке результата. Например, $value в этом примере не помечается:

```
my $value = $tainted ? 'Amelia' : 'Camelia'; # $value не помечается.
```

и в этом тоже:

```
my $value = do {
    if( $tainted ) { 'Amelia' }
    else           { 'Camelia' }
};
```

Поскольку Perl помечает отдельные скаляры, в массиве или хеше одни значения могут оказаться мечеными, а другие – нет. (В хеше мечеными могут быть только значения, но не ключи. Подробнее об этом чуть ниже.)

Следующий код иллюстрирует, как будут работать меченые данные, если выполнить по порядку все эти команды. Команды, отмеченные словом «ненадежна» создают исключительную ситуацию, а те, которые отмечены «OK», – нет.

```
$arg = shift(@ARGV),           # $arg теперь помечена (из-за @ARGV).
$hid = "$arg, 'bar'";          # $hid тоже помечена (из-за $arg).
$line = <>;                    # Помечена (чтение из внешнего файла)
$path = $ENV{PATH};            # Помечена из-за %ENV. но смотрите ниже
$mine = "abc";                 # Не помечена.

system "echo $mine":          # Ненадежна, если не установлена PATH.
system "echo $arg";           # Ненадежна: использует sh с меченой $arg.
system "echo"  $arg;          # OK, если PATH установлена (не использует sh).
system "echo $hid";           # Ненадежна по двум причинам: меченая $hid, PATH

$oldpath = $ENV{PATH};        # $oldpath меченая (из-за %ENV).
$ENV{PATH} = "/bin:/usr/bin", # OK (разрешает выполнять другие программы.)
$newpath = $ENV{PATH};        # $newpath НЕ помечена.

delete @ENV{qw{IFS
               CDPATH
               ENV
               BASH_ENV}};    # Делает %ENV более безопасной.

system "echo $mine",          # OK  безопасна после переустановки PATH
system "echo $hid";           # Ненадежна из-за $hid.

open(OOF, "< $arg");          # OK (open только для чтения не проверяются)
open(OOF, "> $arg");          # Ненадежна (запись в меченый аргумент)

open(OOF, "echo $arg|")       # Ненадежна из-за меченой $arg, но
    || die "невозможность конвейера из echo: $!

open(OOF,"-|")                # Считается OK: см. ниже меченое
    || exec "echo", $arg      # исключение при выполнении списка
    || die "невозможно exec echo: $!";

open(OOF,"-|", "echo", $arg)  # То же, что предыдущее, OK.
    || die " невозможность конвейера из echo: $!";

$shout = `echo $arg`;         # Ненадежна из-за меченой $arg
$shout = `echo abc`;          # $shout помечена из-за обратных апострофов.
$shout2 = `echo $shout`;      # Ненадежна из-за меченой $shout

unlink $mine, $arg;           # Ненадежна из-за меченой $arg
umask $arg;                   # Ненадежна из-за меченой $arg.

exec "echo $arg";             # Ненадежна из-за меченой $arg,
                              # передаваемой интерпретатору команд
exec "echo", $arg;            # Считается OK! (Но смотри ниже.)
exec "sh", "-c", $arg;        # Считается OK, но в действительности не так!
```

Если попытаться сделать что-то небезопасное, возникнет исключительная ситуация (которая, если не перехватывается, становится фатальной), например, "Insecure dependency" или "Insecure $ENV{PATH}". См. далее раздел «Наведение порядка в окружающей среде».

При передаче списка вызову system, exec или open для канала, происхождение аргументов не проверяется, потому что, когда аргументы представлены списком, Perl не требуется вызывать потенциально опасный интерпретатор команд, чтобы выполнить команду. Можно с легкостью написать ненадежные system, exec или open для канала с использованием списочного формата, как показано в последнем примере выше. Эти формы освобождаются от проверки, поскольку предполагается, что вы используете их осознанно.

Иногда, впрочем, невозможно определить, сколько аргументов передается. Передача этим функциям массива[1], содержащего только один элемент, равноценна передаче одной строки, поэтому может быть использован интерпретатор команд. Выход в том, чтобы передать явный путь через косвенные аргументы:

```
system @args;                  # Интерпретатор вызывается только при @args == 1.
system { $args[0] } @args;     # Обходит интерпретатор, даже если один аргумент.
```

Обнаружение и очистка меченых данных

Чтобы проверить, содержит ли скалярная переменная данные сомнительного происхождения, можно обратиться к функции is_tainted, описываемой ниже. Она использует то обстоятельство, что eval *STRING* возбуждает исключение при попытке компиляции данных сомнительного происхождения. Неважно, что переменная $nada, используемая в компилируемом выражении, всегда будет пустой; она все равно окажется меченой, если помечена $arg. Внешняя проверка eval *BLOCK* не производит компиляцию. Ее задача – перехватить исключительную ситуацию, когда внутренняя eval получит меченые данные. Поскольку гарантируется, что переменная $@ не будет пустой после каждого вызова eval, если возникла исключительная ситуация, и будет пустой в противном случае, мы возвращаем результат сравнения длины содержимого этой переменной с нулем:

```
sub is_tainted {
    my $arg = shift;
    my $nada = substr($arg, 0, 0); # нулевая длина
    local $@;                      # сохранить версию вызывающей программы
    eval { eval "# $nada" };
    return length($@) != 0;
}
```

Модуль Scalar::Util, входящий в состав стандартного дистрибутива Perl, уже имеет подобную функцию с именем tainted:

```
use Scalar::Util qw(tainted);

print "Tainted!" if tainted( $ARGV[0] );
```

Модуль Taint::Util из CPAN пошел еще дальше. Он предлагает не только функцию tainted, реализующую то же самое, но и функцию taint, позволяющую пометить любые данные:

---

[1] Или функции, создающей список.

```
use Taint::Util qw(tainted taint);

my $scalar = 'This is untainted';  # непомеченная
taint( $scalar );                  # теперь меченая
```

Это может пригодиться для тестирования сценариев с мечеными данными:

```
use Test::More;
use Taint::Util qw(tainted taint);

my $tainted = 'This is untainted'; # непомеченная
taint( $tainted );                 # теперь меченая

ok( tainted( $tainted ), 'Data are tainted' );
is( refuse_to_work( $tainted ), undef, 'Returns undef with tainted data' );

done_testing();
```

Но проверка происхождения данных – лишь часть дела. Как правило, мы прекрасно знаем, какие переменные содержат меченые данные, и надо лишь сбросить признак «помеченности». Единственный официальный путь обхода механизма проверки меченых данных – это ссылка на вложенные соответствия, полученные ранее операцией сопоставления с регулярным выражением.[1] Если шаблон содержит сохраняющие скобки, к найденным подстрокам можно обращаться через переменные $1, $2 и $+, либо применив шаблон в списочном контексте. В любом случае предполагается, что писали шаблон осознанно, и написали его так, чтобы вычистить все, представляющее опасность. Поэтому следует крепко подумать, прежде чем просто снять «помеченность», иначе вы нарушите работу этого механизма.

Лучше убедиться, что переменная содержит только «хорошие» символы, чем проверять наличие «плохих», потому что последние слишком легко пропустить, если вы о них не подумали. Следующий тест проверяет, не содержит ли $string что-нибудь, кроме «символов слов» (букв, цифр и подчеркиваний), дефисов, знаков @ и точек:

```
if ($string =~ /^([-\@\w.]+)$/) {
    $string = $1; # $string теперь не помечена.
}
else {
    die "Плохие данные в $string"; # Записать куда-то в журнал.
}
```

В результате $string становится достаточно безопасной для дальнейшего использования во внешней команде, так как /\w+/ обычно не соответствует метасимволам интерпретатора команд, да и другие символы не должны означать для него

---

[1] Неофициальный путь состоит в том, чтобы сохранить меченую строку как ключ хеша и получить этот ключ обратно. Поскольку ключи не являются в действительности полноценными SV (внутреннее имя для scalar values – скалярных значений), у них отсутствует свойство «помеченности». Однако такое поведение может в один прекрасный день поменяться, поэтому не следует на него полагаться. Будьте осторожны при работе с ключами, чтобы не снять непреднамеренно «помеченность» со своих данных и не сделать с ними что-нибудь небезопасное.

чего-либо особенного.[1] Если бы мы использовали /(.+)/s, это было бы небезопасно, поскольку такой шаблон пропускает все. Но Perl это не проверяет. При снятии «помеченности» будьте предельно осторожны со своими шаблонами. Очистка данных с помощью регулярных выражений является *единственным* одобряемым механизмом снятия метки с «грязных» данных. А иногда это оказывается совершенно неверным подходом. Если сценарий оказывается в режиме проверки меченых данных из-за запуска с повышенными привилегиями, а не потому, что был указан ключ *-T*, уменьшить риск можно, породив процесс (fork a child) с пониженными привилегиями; см. раздел «Наведение порядка в окружающей среде».

Прагма use re 'taint' запрещает неявное снятие меток при поиске по шаблону до конца текущей лексической области видимости. Ее можно использовать, чтобы извлечь какие-то подстроки из потенциально меченых данных, и оставить эти подстроки мечеными, чтобы отложить принятие мер по защите на потом.

Предположим, что вы осуществляете такой поиск, причем $fullpath является меченой:

```
($dir, $file) = $fullpath =~ m!(.*/)(.*)!s;
```

По умолчанию $dir и $file теперь не будут помечены. Но, вероятно, вы не стали бы делать это столь бесцеремонно, если бы задумались о проблемах безопасности. Например, было бы не слишком радостно получить в $file строку "; rm -rf * ;", что можно привести в качестве довольно вопиющего примера. В следующем примере две переменные с результатами остаются мечеными, если помечена $fullpath:

```
{
    use re "taint";
    ($dir, $file) = $fullpath =~ m!(.*/)(.*)!s;
}
```

Разумно по умолчанию оставлять вложенные соответствия мечеными по всему исходному файлу, и лишь при необходимости избирательно разрешать снятие меток во вложенных областях видимости:

```
use re "taint";
# в остальной части файла $1 и другие остаются мечеными
{
    no re "taint";
    # в этом блоке снимаются метки при поиске регулярных выражений
    if ($num =~ /^(\d+)$/) {
        $num = $1;
    }
}
```

Данные, полученные из дескриптора файла или каталога, помечаются автоматически, за исключением случая, когда они извлекаются из особого дескриптора DATA. При желании другие дескрипторы тоже можно пометить, как заслуживающие доверия, с помощью функции untaint из модуля IO::Handle:

[1] Если только вы не используете специально искаженные национальные настройки. Perl допускает возможность компрометации национальных настроек системы. Поэтому при выполнении с директивой use locale шаблоны, содержащие классы, такие как \w или [[:alpha:]], дают недостоверные результаты.

```
use IO::Handle;

IO::Handle::untaint(*SOME_FH); # Либо как процедуру,
SOME_FH->untaint();            # либо в стиле ООП
```

Отключение пометки данных для дескриптора в целом является рискованным шагом. Откуда вы можете знать, что он *действительно* безопасен? Если вы собираетесь это сделать, то должны хотя бы проверить, что никто, кроме владельца, не может осуществлять запись в этот файл.[1] Если вы работаете с файловой системой UNIX (в которой вызов *chown*(2) благоразумно разрешен только суперпользователю), можно использовать такой код:

```
use File::stat;
use Symbol "qualify_to_ref";
sub handle_looks_safe(*) {
    my $fh = qualify_to_ref(shift, caller);
    my $info = stat($fh);
    return unless $info;

    # владельцем не являются ни суперпользователь, ни "я",
    # чей реальный uid находится в переменной $<
    if ($info->uid != 0 && $info->uid != $<) {
        return 0;
    }

    # проверить, могут ли писать в файл группа или другие.
    # использовать также 066 для определения возможности чтения
    if ($info->mode & 022) {
        return 0;
    }
    return 1;
}

use IO::Handle;
SOME_FH->untaint() if handle_looks_safe(*SOME_FH)
```

Мы вызывали stat для дескриптора файла, а не его имени, чтобы избежать опасной ситуации, в которой велика вероятность взаимоблокировок. См. раздел «Обработка состояния гонки» далее в этой главе.

Заметим, что эта программа служит лишь хорошим началом. Несколько более параноидальная версия проверила бы еще все родительские каталоги, даже несмотря на то, что применение stat к дескрипторам каталогов не дает надежных результатов. Но если в какой-нибудь родительский каталог может осуществляться запись кем угодно, знайте – вы в опасности, независимо от того, возникает ли состояние гонки (race condition).

Perl имеет собственное представление о том, какие операции опасны, но нажить неприятностей можно через другие операции, которые не анализируют свои аргументы. Не всегда можно ограничиться проверкой входных данных. Функции вывода Perl не проверяют, являются ли мечеными их аргументы, но в некоторых

---

[1] С дескриптора каталога тоже можно снять признак мечености, но конкретно эта функция работает только с дескриптором файла. Это связано с тем, что в случае дескриптора каталога не существует переносимого способа передать его указатель файла в stat.

окружениях это существенно. Если не следить за тем, что выводится, получатель, обрабатывающий эти данные, может столкнуться с неожиданными эффектами обработки. Если вы работаете с терминалом, некоторые управляющие символы и коды могут вызвать такое поведение терминала, которое озадачит пользователя. В среде веб-сервера бездумный возврат полученных из другого источника данных может привести к непреднамеренному внедрению в страницу тегов HTML, которые кардинально изменят ее внешний вид. Хуже того, некоторые теги способны привести к выполнению кода в браузере.

Представьте типичную гостевую книгу, где пользователи оставляют свои сообщения, доступные другим посетителям. Гость-злоумышленник может передать специальные теги HTML или вставить теги `<SCRIPT>...</SCRIPT>`, которые выполнят код (например, JavaScript) в браузерах других посетителей.

Следует проявлять такую же осторожность в среде веб при выводе данных, полученных от пользователя, как при проверке на отсутствие недопустимых символов при просмотре меченых данных, обращающихся к ресурсам вашей собственной системы. Например, чтобы удалить все символы, не входящие в список одобренных, попробуйте конструкцию вроде этой:

```
$new_guestbook_entry =~ tr[_a-zA-Z0-9 ,./!?()@+*-][]dc;
```

Конечно, не следует использовать это для обработки имени файла: как минимум, вам едва ли нужны имена файлов с пробелами и косыми. Но, чтобы в вашу гостевую книгу не прокрались теги и объекты HTML, этого достаточно. В каждой ситуации очистка данных имеет свои особенности, поэтому всегда отводите немного времени на то, чтобы определить допустимые и недопустимые данные. Механизм проверки меченых данных служит для перехвата глупых ошибок и не отменяет необходимости думать.

Наведение порядка в окружающей среде

При запуске из сценария на Perl другой программы – неважно, каким образом, – Perl проверяет, является ли безопасной переменная среды `PATH`. Поскольку `PATH` происходит из системного окружения, она сразу помечается, и при попытке запустить другую программу Perl возбуждает исключение `"Insecure $ENV{PATH}"`. Если установить `PATH` равной известному немеченому значению, Perl убедится, что все перечисленные в `PATH` каталоги не доступны для записи никому, кроме владельца и группы каталога; в противном случае Perl возбудит исключение `"Insecure directory"`.

Вас может удивить, что Perl интересуется вашей переменной `PATH`, даже если вы задаете полное имя команды, которую вы хотите выполнить. Верно, что, если указано абсолютное имя, `PATH` не используется для поиска выполняемого модуля. Но вы не можете быть уверены, что запускаемая программа не развернется на 180 градусов и не запустит *другую* программу, с которой возникнут неприятности из-за незащищенной `PATH`. Поэтому Perl требует установить надежное значение `PATH`, прежде чем вызывать какую-либо программу каким бы то ни было способом.

`PATH` – не единственная переменная среды, способная доставить огорчения. Поскольку некоторые интерпретаторы команд используют переменные `IFS`, `CDPATH`, `ENV` и `BASH_ENV`, Perl проверяет, что все они пусты либо не помечены перед тем, как запустить другую команду. Либо назначьте этим переменным безопасные значения, либо вообще удалите из окружения:

```
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)}; # Сделайте %ENV безопаснее
```

Возможности системы, удобные в обычной среде, могут стать проблемой для защиты данных во враждебной среде. Даже если вы не забываете запрещать имена файлов, содержащие символы перевода строки, важно помнить, что open обращается не только к именованным файлам. Если надлежащим образом декорировать аргумент с именем файла, вызовы open с одним или двумя аргументами могут запускать произвольные внешние программы через каналы, ответвлять дополнительные экземпляры текущего процесса, дублировать дескрипторы файлов и интерпретировать специальное имя файла «-» как псевдоним стандартного ввода или вывода. Они могут также игнорировать ведущие и замыкающие пробельные символы, что скроет такие необычные аргументы от ваших проверочных шаблонов. Механизм проверки меченых данных в Perl действительно перехватывает меченые аргументы, используемые при открытии конвейеров (если только не используется отдельный список аргументов) и файлов (если файл открывается не только для чтения), но возбуждаемые при этом исключения могут нарушить нормальное выполнение вашей программы.

Если в качестве имени открываемого файла вы собираетесь использовать данные, полученные извне, хотя бы включите явно через пробел режим открытия. Однако безопаснее всего, по-видимому, использовать низкоуровневую sysopen или форму open с тремя аргументами:

```
# Волшебное открытие - может быть любым
open(FH, $file)         || die "невозможно волшебно открыть $file: $!";

# Гарантирует открытие файла только для чтения и исключает открытие канала
# или запуск порожденного процесса, но все же принимает дескрипторы файлов и "-",
# и игнорирует пробельные символы с обоих концов имени.
open(FH, "< $file")     || die "невозможно открыть $file: $!";

# WYSIWYG-открытие: отключает все удобные функции
open(FH, "<", $file)    || die "невозможно открыть $file: $!";

# Те же свойства, что в 3-аргументной версии WYSIWYG.
require Fcntl;
sysopen(FH, $file, O_RDONLY) || die "невозможно sysopen $file: $!";
```

Даже этих шагов не вполне достаточно. Perl не препятствует открытию файлов с мечеными именами в режиме чтения, поэтому следует проявлять осторожность, показывая их содержимое людям. Программа, которая открывает файл с произвольным именем, указанным пользователем, и показывает его содержимое, представляет собой угрозу безопасности. А если это личное письмо? А если это системный файл с паролями? А если это данные о зарплате или о вашем портфеле ценных бумаг?

Изучите внимательно имена файлов, полученные от потенциально враждебного пользователя,[1] прежде чем открывать их. Например, можно проверить отсутствие в пути подлых составляющих каталогов. Известными трюками такого сорта являются имена типа «*../../../../../../../etc/passwd*». Защититься можно, проверив отсутствие косой черты в имени (если она служит разделителем каталогов в вашей

---

[1] А в Сети единственными пользователями, которых вы можете не считать потенциально враждебными, являются активно враждебные пользователи.

системе). Часто также используется прием включения символов перевода строки или точки с запятой в имя файла, в результате чего какой-нибудь несчастный глупый интерпретатор командной строки может принять их за начало новой команды посреди имени файла. Именно поэтому в режиме проверки меченых данных отвергаются непроверенные внешние команды.

Доступ к командам и файлам с ограниченными привилегиями

Последующее изложение касается некоторых ловких приемов, относящихся к безопасности в UNIX-подобных системах. Пользователи других систем могут спокойно пропустить этот раздел (хотя навряд ли это будет безопасно).

Если сценарий запущен как set-id, постарайтесь по возможности выполнять опасные операции с правами пользователя, а не программы. То есть, собираясь применить open, sysopen, system, обратные апострофы или другие операции с файлами или процессами, защитите себя, установив текущие UID или GID равные реальным UID или GID. В сценариях setuid это можно сделать, сказав `$> = $<` (или `$EUID = $UID` в области действия прагмы use English), а в сценариях setgid, сказав `$( = $)` (`$EGID = $GID`). Если установлены оба идентификатора, следует изменить оба. Однако иногда это невозможно, поскольку расширенные права доступа могут все же понадобиться вашей программе позже.

Для таких случаев Perl поддерживает достаточно надежный способ открытия файла или канала из программы set-id. Во-первых, следует породить процесс с помощью особого синтаксиса open, связывающего родительский и порожденный процессы через канал. Во-вторых, в порожденном процессе установить ID пользователя и группы в исходные или заведомо безопасные значения. Можно также изменить атрибуты порожденного процесса, не затрагивая родительский, что позволит изменить рабочий каталог, установить маску создания файла и переменные среды. Наконец, лишившись дополнительных прав доступа, порожденный процесс вызывает open и передает любые данные, доступные от имени простого, но слабоумного пользователя, своему могущественному, но по справедливости параноидальному родителю.

В то время как system и exec не используют интерпретатор команд, получив более одного аргумента, оператор обратных апострофов не имеет такого альтернативного соглашения по вызову. С помощью приема ветвления легко можно эмулировать обратные апострофы, не опасаясь их обработки интерпретатором команд, и с пониженными (а потому более безопасными) правами доступа:

```
use English, # чтобы использовать $UID и прочие
die "Невозможно породить через open: $!" unless defined($pid = open(FROMKID, "-|"));
if ($pid) {  # родитель
    while (<FROMKID>) {
        # какие-то действия
    }
    close FROMKID;
}
else {
    $EUID = $UID; # setuid(getuid())
    $EGID = $GID; # setgid(getgid()), и initgroups(2) по getgroups(2)
    chdir("/")     || die "невозможно chdir в /: $!";
```

```
        umask(077);
        $ENV{PATH} = "/bin:/usr/bin";
        exec "myprog", "arg1", "arg2";
        die "невозможно exec myprog: $!",
    }
```

Этот способ вызова программ из сценария set-id превосходит все остальные. Он гарантирует, что интерпретатор команд не будет использоваться для выполнения чего-либо и обеспечит понижение привилегий перед запуском программы. (Но из-за того, что списочные формы system, exec, а также форма open, открывающая канал, не подвергаются проверке чистоты аргументов, все же следует проявлять осторожность в отношении того, что вы передаете.)

Если потребуется сохранить права доступа и реализовать обратный апостроф или открытие канала, не опасаясь перехвата интерпретатором команд ваших аргументов, можно использовать такой код:

```
open(FROMKID, "-|") || exec("myprog", "arg1", "arg2")
    || die "невозможно выполнить myprog: $!";
```

а затем просто читать из FROMKID в родительском процессе. Начиная с Perl версии 5.6.1, это можно выразить так:

```
open(FROMKID, "-|", "myprog", "arg1", "arg2");
```

Прием ветвления удобно использовать не только для запуска команд из программ set-id. Его можно применять и для открытия файлов с правами пользователя, который запустил программу. Допустим, что у вас есть программа setuid, которой нужно открыть файл для записи. Вы не хотите выполнять open с дополнительными правами доступа, но окончательно потерять их вы тоже не можете. В таком случае запустите порожденную копию, которая отбросит дополнительные права и выполнит open в пользу основной программы. Если потребуется выполнить запись в файл, передайте данные порожденному процессу, а уже он запишет их в файл.

```
use English;

defined ($pid = open(SAFE_WRITER, "|-"))
    || die "Невозможно ветвление: $!";

if ($pid) {
    # родитель, передать данные дочернему процессу через SAFE_WRITER
    print SAFE_WRITER "@output_data\n";
    close SAFE_WRITER
        || die $! ? "Syserr при закрытии SAFE_WRITER: $!"
                  : "Статус ожидания $? от SAFE_WRITER";
}
else {
    # дочерний процесс, отменить дополнительные права
    ($EUID, $EGID) = ($UID, $GID);

    # открыть файл с правами исходного пользователя
    open(FH, "> /some/file/path")
        || die "невозможно открыть /some/file/path для записи: $!";

    # копировать от родителя (теперь это stdin) в файл
```

```
        while (<STDIN>) {
            print FH $_;
        }
        close(FH)     || die "ошибка при закрытии: $!";
        exit;         # Не забудьте сделать так, чтобы SAFE_WRITER исчез.
    }
```

Если файл открыть невозможно, порожденный процесс выведет сообщение об ошибке и завершит работу. Если родительский процесс попытается вывести данные в недействительный теперь дескриптор файла в порожденном процессе, то получит сигнал разрушения канала (SIGPIPE), который фатален, если не перехватывается или игнорируется. См. раздел «Сигналы» в главе 15.

Обход режима проверки меченых данных

Режим проверки меченых данных – это инструмент разработчика, помогающий обнаруживать данные, требующие очистки. Он не является абсолютной гарантией от неприятностей, поэтому неприятности все же возможны. Фактически этот режим можно легко обойти.

Ключ командной строки *-T* обеспечивает принудительную проверку меченых данных, и его можно добавить в строку #! в вашем сценарии:

```
#!/usr/bin/perl -T

system 'echo', $ARGV[0];
```

Если попытаться запустить такой сценарий из командной строки без ключа *-T*, он завершится ошибкой:

```
% perl echo.pl
"-T" is on the #! line, it must also be used on the command line
(ключ "-T" присутствует в строке #!, он также должен быть указан в командной строке)
```

Коварный пользователь может включить режим проверки меченых данных, но превратить обычно фатальные сообщения в предупреждения. Ключ *-t* включает режим проверки меченых данных, но в этом случае при нарушениях режима будут выводиться только предупреждения. Система сможет принимать меченые данные:

```
% perl -t echo.pl Amelia
Insecure $ENV{PATH} while running with -t switch
Insecure dependency in system while running with -t switcn
Insecure $ENV{PATH} while running with -t switch
Amelia
```

Если запускается сценарий setuid и режим проверки меченых данных включается автоматически, его также можно обойти с помощью ключа *-t*:

```
% perl -t echo.pl Amelia
Insecure $ENV{PATH} while running with -t switch
Insecure dependency in system while running with -t switch
Insecure $ENV{PATH} while running with -t switch
Amelia
```

Аналогично, ключ *-U* разрешает *perl* выполнять «небезопасные» операции, но при этом все еще требуется указать ключ *-T*:

```
% perl -TU echo.pl Amelia
Amelia
```

Если потребуется вернуть предупреждения на место, используйте *-w*:

```
% perl -TU -w echo.pl Amelia
Insecure $ENV{PATH} while running with -t switch
Insecure dependency in system while running with -t switch
Insecure $ENV{PATH} while running with -t switch
Amelia
```

Программисты способны обойти режим проверки меченых данных, но не приемы по очистке данных, которые были показаны выше. Например, ниже представлена операция сопоставления, совпадающая со всем чем угодно:

```
my $untainted = $tainted =~ m/(.*)/;
```

Ее легко заметить, читая исходный текст программы, поэтому коварный пользователь сумеет передать данные через хеш. Поскольку понятие «меченых данных» применяется к скалярным *переменным*, ключи хеша не будут считаться мечеными. Поэтому использование скалярной переменной в качестве ключа хеша помогает устранить все волшебство:

```
my (untainted) = keys %{ { $untainted => 1 } };
```

О подобных приемах можно рассказывать часами. Другие ухищрения можно найти в главе «Secure Programming Techniques» книги «Mastering Perl».

Обработка ошибок синхронизации

Иногда работа вашей программы очень чувствительна к неподвластному вам расписанию внешних событий. Это всегда проблема, когда другие программы, в особенности враждебные, соперничают с вашей программой за одни и те же ресурсы (например, файлы или устройства). В многозадачной среде невозможно предсказать порядок, в котором процессы, ожидающие своей очереди выполнения, получат доступ к процессору. Потоки команд возможных процессов чередуются, поэтому сначала один процесс получает несколько циклов центрального процессора, затем другой процесс и т.д. Очередность выполнения и отводимое на него время кажутся случайными. Если программа одна, то проблем не возникает, в отличие от ситуации, когда несколько программ совместно используют общие ресурсы.

Особенно чувствительны к таким вопросам программисты, создающие многопоточные программы. Они быстро усваивают, что нельзя говорить:

```
$var++ if $var == 0;
```

когда нужно сказать:

```
{
    lock($var);
    $var++ if $var == 0;
}
```

В первом случае получаются непредсказуемые результаты, если несколько потоков попытаются одновременно выполнить этот код. Если совместно используемыми объектами являются файлы, и процессы, подобно потокам, соперничают

за доступ к ним, возникают такие же проблемы. Процесс, в конце концов, – это в некотором отношении лишь поток. Или наоборот.

Непредсказуемость во времени влияет на любые ситуации, привилегированные и непривилегированные. Сначала рассмотрим, как справиться с давней ошибкой в старых ядрах UNIX, проявляющейся в программах set-id. Затем перейдем к обсуждению состояний гонок (race conditions) в целом, того, как они могут превращаться в бреши в системе защиты, и действий, помогающих этого избежать.

Ошибки защиты в ядре UNIX

Помимо очевидных проблем из-за особых прав, что даются таким гибким и непроницаемым интерпретаторам, какими являются интерпретаторы команд, в старых версиях UNIX в ядре имеется ошибка, которая делает небезопасными все сценарии set-id еще до того, как они попадут интерпретатору. Проблема состоит не в самом сценарии, а в ситуации, возникающей при обнаружении ядром сценария set-id для выполнения. (Этой ошибки нет в системах, ядро которых не распознает #!.) Когда ядро открывает такой файл, чтобы узнать, какой интерпретатор нужно запустить, возникает некоторая задержка, прежде чем запустится интерпретатор (теперь set-id) и откроет этот файл. Эта задержка дает возможность злоумышленнику изменить файл, особенно если система поддерживает символические ссылки.

К счастью, иногда эту «возможность» ядра можно отключить. К несчастью, сделать это можно разными способами. Система может запретить сценарии с установленными битами set-id, что не очень удобно. Кроме того, она может игнорировать биты set-id у сценариев. В последнем случае Perl может эмулировать механизм setuid и setgid, обнаружив биты set-id (в другом отношении бесполезные) у сценариев Perl. Делает он это с помощью особого исполняемого модуля *suidperl*, который при необходимости вызывается автоматически.[1] Однако если в ядре *не* отключена функция сценариев set-id, Perl начинает громко жаловаться, что данный сценарий setuid небезопасен. Придется либо отключить в ядре поддержку сценариев set-id, либо поместить сценарий в C-обертку. C-обертка – это просто скомпилированная программа, единственным действием которой является вызов вашей программы на Perl. Скомпилированные программы не подвержены действию той ошибки ядра, от которой страдают сценарии set-id.

Вот простая обертка, написанная на C:

```
#define REAL_FILE "/path/to/script"
main(ac, av)
    char **av;
{
    execv(REAL_FILE, av);
}
```

Скомпилируйте ее в исполняемый образ и запускайте *вместо* своего сценария set-id. Используйте абсолютное имя файла, так как C недостаточно сообразителен, чтобы проверять меченые данные по вашей переменной PATH.

---

1 При необходимости *и* разрешении на это. Если Perl обнаружит, что файловая система, в которой находится сценарий, смонтирована с параметром nosuid, это значение будет соблюдаться. Здесь нельзя использовать Perl, чтобы обойти политику защиты данных вашего системного администратора.

Другой возможный подход состоит в применении экспериментального генератора кода C в компиляторе Perl. В скомпилированном образе вашего сценария вероятность состояния гонки исключается (см. главу 16).

В последние годы наконец стали поставляться системы с исправленной ошибкой set-id. В этих системах при передаче интерпретатору имени сценария set-id больше не используется имя файла, которое можно подделать, а передается особый файл, представляющий дескриптор файла, например */dev/fd/3*. Этот особый файл уже открыт для сценария, поэтому состояния гонки не могут возникать и, соответственно, использоваться злонамеренными сценариями.[1] Самые последние версии UNIX используют такой подход, чтобы избежать состояния гонки, возникающего при открытии одного и того же имени файла дважды.

Обработка состояний гонки

Вот мы и подошли к теме *состояний гонок* (*race conditions*). Что же это за состояния? Они часто всплывают при обсуждении проблем защиты данных. (Хотя и реже, чем в небезопасных программах. К несчастью.) Дело в том, что они служат обильным источником тонких программных ошибок, а такие ошибки часто становятся основой *подвигов* на ниве проникновения (если вежливо назвать таким словом взлом системы защиты данных). Возникает состояние гонки, когда результат нескольких взаимосвязанных событий зависит от порядка их возникновения, но этот порядок нельзя гарантировать из-за недетерминированных временных эффектов. Каждое событие стремится осуществиться первым, а о конечном состоянии системы можно только гадать.

Представьте, что имеется один процесс, который перезаписывает существующий файл, и другой процесс, который читает тот же самый файл. Нельзя предсказать, что вы прочтете: старый файл, новый файл или случайную смесь их обоих. Неизвестно даже, все ли данные будут прочитаны. Читающая программа может выиграть гонку, прочесть файл до конца и завершить работу. Тем временем пишущая программа продолжит работу уже после того, как читающая программа нашла конец файла, файл продолжится далее того места, где было прекращено его чтение, а читающая программа об этом не узнает.

В данном случае решение простое: обе стороны должны заблокировать файл с помощью `flock`. Обычно читающая программа запрашивает блокировку с совместным доступом, а пишущая – с монопольным. Если все стороны запрашивают эти рекомендованные блокировки и соблюдают их, то чтение и запись не будут перемежаться и данные не исказятся. См. раздел «Блокировка файлов» главы 15.

Менее очевидная форма состояния гонки может возникнуть, когда операции над именем файла управляют последующими операциями над этим файлом. Когда операторы проверки файлов осуществляются с именами файлов, а не с их дескрипторами, это прямая дорога к возникновению состояния гонки. Рассмотрим такой код:

```
if (-e $file) {
    open(FH, "<", $file)
```

[1] В этих системах нужно компилировать Perl с ключом *-DSETUID_SCRIPTS_ARE_SECURE_NOW*. Программа *Configure*, которая собирает Perl, старается самостоятельно определить потребность в этом ключе, чтобы вам не пришлось этим заниматься.

```
        || die "невозможно открыть $file для чтения: $!";
}
else {
    open(FH, ">", $file)
        || die "невозможно открыть $file для записи: $!";
}
```

Этот код кажется тривиальным и, тем не менее, может вызвать состояние гонки. Нет никакой гарантии, что результат выполнения проверки -e все еще будет действителен в том или ином вызове open. Другой процесс может удалить файл, прежде чем удастся его открыть в блоке if, и файл, который, как вам казалось, должен быть на месте, найден не будет. Другой процесс может создать файл, прежде чем второй вызов open в блоке else получит возможность создать файл, и файл, которого, как вам казалось, не существует, окажется там. Простая функция open создает новые файлы, но перезаписывает существующие. Допустим, вы решили перезаписать некоторый существующий файл, но что если этот файл является вновь созданным псевдонимом или символической ссылкой на другой файл в системе, который вам очень не хотелось бы перезаписать? Можно считать, что значение каждого имени файла в каждый данный момент известно, но в действительности в этом никогда нельзя быть уверенным, если в системе выполняются другие процессы с правом доступа к каталогу, содержащему этот файл.

Чтобы решить описанную проблему, нужно использовать функцию sysopen, которая позволяет отдельно управлять тем, нужно ли создавать новый файл и можно ли разрушить существующий. И мы откажемся от проверки существования файла -e, поскольку она не приносит здесь никакой пользы, а только увеличивает шансы попасть в состояние гонки.

```
use Fcntl qw/O_WRONLY O_CREAT O_EXCL/;
open(FH, "<", $file)
    || sysopen(FH, $file, O_WRONLY | O_CREAT | O_EXCL)
    || die "невозможно создать новый файл $file: $!";
```

Теперь, даже если файл каким-то образом возникнет в промежутке времени между отказом open и попыткой sysopen открыть новый файл для записи, никакого вреда не будет, потому что с данными флагами sysopen не станет открывать существующий файл.

Если кто-то попытается заставить вашу программу дурно себя вести, весьма вероятно, что он будет делать это, создавая и уничтожая файлы, когда вы этого не ожидаете. Один из способов снизить риск быть обманутым состоит в том, чтобы дать себе слово никогда не выполнять операции над файлом с одним и тем же именем больше одного раза. Как только вы открыли файл, забудьте его имя (вспоминая, может быть, только в сообщениях об ошибках) и работайте только с дескриптором, представляющим файл. Это значительно безопаснее, так как даже если кто-то сможет манипулировать именами ваших файлов, делать это с дескрипторами он не сумеет. (Только если вы дадите ему разрешение – см. раздел «Передача дескрипторов файлов» в главе 15.)

Ранее в этой главе мы продемонстрировали функцию handle_looks_safe, которая вызывала функцию Perl stat с дескриптором (не именем) файла для проверки владельца и прав доступа к файлу. Использование дескриптора файла важно для корректности: если бы мы работали с именем файла, не было бы никакой гарантии, что файл, атрибуты которого исследовались, тот же самый, что и только что

открытый (или который собирались открыть). Какой-нибудь злоумышленник мог удалить файл и быстро подменить его своим с какой-нибудь гадостью в некий момент между вызовами stat и open. Неважно, которая из них вызывается первой; возможность нечестной игры в перерыве между вызовами в любом случае сохраняется. Кто-то может счесть риск пренебрежимо малым, ведь промежуток времени так короток, однако существует множество сценариев взлома, которые будут рады запустить вашу программу тысячу раз, чтобы поймать единственный случай, когда она окажется недостаточно осторожной. Толковый сценарий взлома может даже понизить приоритет программы, чтобы она прерывалась чаще, и несколько ускорить события. Люди напряженно работают над такими вещами, поэтому их и называют *подвигами* (*exploits*).

Вызывая stat с уже открытым дескриптором файла, мы используем имя файла только один раз и поэтому избегаем состояния гонки. Хорошей стратегией для устранения гонки между двумя событиями является их объединение в одну операцию атомарного характера.[1] Поскольку мы обращаемся к файлу по имени только один раз, ситуация гонки в промежутках между несколькими обращениями не возникает, поэтому не важно, изменится ли имя. Даже если взломщик удалит открытый нами файл (да, бывает и такое) и подменит его другим, чтобы обмануть нас, мы сохраним дескриптор исходного файла.

Временные файлы

Помимо выходов за пределы буферов (которым сценарии Perl практически не подвержены) и доверия не заслуживающим того входным данным (защитой от чего служит режим проверки меченых данных), некорректное создание временных файлов является одной из наиболее часто используемых брешей в системе защиты. К счастью, атака на временные файлы обычно требует от взломщиков наличия законной учетной записи пользователя, что существенно сокращает число потенциальных негодяев.

Часто по небрежности программы используют временные файлы различными небезопасными способами, например помещают их в каталоги, доступные для записи кому угодно, используют легко угадываемые имена и не проверяют существование файла с тем же именем. Если вы обнаружите в программе такой код:

```
open(TMP, "> /tmp/foo.$$")
    || die "can't open /tmp/foo.$$: $!";
```

знайте, что столкнулись сразу со всеми тремя перечисленными ошибками. Эта программа просто напрашивается на неприятности.

Взломщик может подбросить файл с тем же именем, которое вы собираетесь использовать. Добавление PID процесса в конец имени не является достаточным для уникальности – как это ни удивительно, угадать PID совсем не трудно.[2] В ре-

---

[1] Да, операции над атомами можно проводить и в безъядерной зоне. Когда Демокрит назвал «атомами» невидимые частицы вещества, он буквально имел в виду нечто неделимое: ἀ- (не) + -τομος (делимый). Атомарная операция – это действие, которое нельзя прервать. (Попробуйте как-нибудь прервать атомную бомбу.)

[2] Если только у вас не система типа OpenBSD, которая назначает новые PID случайным образом.

зультате программа с неосторожным вызовом open уже не создаст временный файл для собственных целей, а перезапишет файл взломщика.

Так какой вред это может причинить? Разнообразный. Файл взломщика может оказаться не простым файлом, а символической ссылкой (а иногда и жесткой ссылкой), указывающей на некоторый важный файл, недоступный взломщику для записи, например */etc/passwd*. Программа считает, что открыла в */tmp* совершенно новый файл, но на самом деле она разрушила существующий файл в каком-то другом месте.

В Perl есть две функции, которые при правильном применении помогут решить эти проблемы. Первая из них – POSIX::tmpnam, возвращающая имя открываемого файла:

```
# Перебирать имена, пока не получим совершенно новое.
use POSIX;
do {
    $name = tmpnam(),
} until sysopen(TMP, $name, O_RDWR | O_CREAT | O_EXCL, 0600);
# Теперь можно осуществлять ввод/вывод с помощью дескриптора TMP
```

Вторая – IO::File::new_tmpfile, возвращающая дескриптор уже открытого файла:

```
# Или позволить модулю сделать это вместо нас.
use IO::File;
my $fh = IO::File::new_tmpfile(); # это POSIX-функция tmpfile(3)
# Теперь можно осуществлять ввод/вывод с помощью дескриптора $fh
```

Ни тот ни другой методы не совершенны, но первый предпочтителен. Основной недостаток второго способа заключается в том, что Perl зависит от слабостей реализации *tmpfile*(3) в системной библиотеке C, и нет никаких гарантий, что эта функция не делает чего-либо столь же опасного, как open, которую мы пытаемся исправить. (А некоторые, к сожалению, делают.) Менее существенная проблема в том, что эта функция вообще не возвращает имя файла. Хотя лучше будет, если вы сможете работать с временным файлом без имени, поскольку в результате вы не спровоцируете состояние гонки его повторным открытием, но часто без имени работать невозможно.

Первый способ плох главным образом тем, что не позволяет указать каталог для размещения временного файла, как функция *mkstemp*(3) из библиотеки C. Во-первых, крайне нежелательно, чтобы файл оказался в сетевой файловой системе. Флаг O_EXCL не гарантирует правильной работы в сетевой файловой системе, поэтому несколько процессов, запросивших монопольное создание примерно в одно время, могут коллективно преуспеть в этом. Во-вторых, поскольку каталог может оказаться доступным для записи другим, у злоумышленника есть возможность подкинуть символическую ссылку на несуществующий файл, что заставит программу создать файл в месте, выбранном не вами.[1] Если необходимо что-то сохранить во временных файлах, не помещайте их в каталог, доступный для записи всем. При необходимости используйте флаг O_EXCL в sysopen и постарайтесь работать с каталогами, для которых установлен флаг «удаляет только владелец» – так называемый «липкий» (sticky) бит.

---

[1] Решением в некоторых операционных системах является вызов sysopen с флагом O_NOFOLLOW. Это вызывает неудачу выполнения функции, если конечная составляющая пути является символической ссылкой.

В Perl версии 5.6.1 появился третий путь. Стандартный модуль File::Temp учитывает все сложности, о которых мы говорили. Например, можно использовать параметры по умолчанию:

```
use File::Temp "tempfile";
$handle = tempfile();
```

Или определить некоторые из них:

```
use File::Temp "tempfile";
($handle, $filename) = tempfile("plughXXXXXX",
                                DIR => "/var/spool/adventure",
                                SUFFIX = ".dat");
```

Модуль File::Temp предоставляет также безопасные версии других упомянутых нами функций (хотя родной интерфейс лучше, поскольку возвращает открытый дескриптор файла, а не просто имя файла, подверженное состоянию гонки). См. более подробное описание параметров и семантики этого модуля.

Получив дескриптор файла, с ним можно делать что угодно. Он открыт для чтения и записи, поэтому можно писать в дескриптор, перемещаться в начало файла и при необходимости перезаписывать то, что было только что записано или прочитано. Но чего *действительно* нужно избегать, так это повторного открытия того же файла, потому что нельзя быть уверенным, что это тот же самый файл, который был открыт в первый раз.[1]

Когда вы запускаете из своего сценария другую программу, Perl обычно закрывает все дескрипторы файлов, чтобы прикрыть другую брешь в системе безопасности. Если вы используете fcntl, чтобы сбросить флаг закрытия при exec (как показано в конце описания open в главе 27), другие вызываемые программы наследуют этот новый дескриптор открытого файла. В системах, поддерживающих каталог */dev/fd/*, можно передать другой программе имя файла, в действительности означающее дескриптор файла, построив его так:

```
$virtname = "/dev/fd/"   fileno(TMP);
```

Если необходимо только вызвать подпрограмму или программу Perl, которые рассчитывают получить в качестве аргумента имя файла, и вы знаете, что эта подпрограмма или программа вызывают для его открытия обычную функцию open, можно передать дескриптор с использованием обозначений Perl для дескриптора файла:

```
$virtname = "=&" . fileno(TMP);
```

Если такое «имя» передать обычной функции open с одним или двумя аргументами (но не тремя; третий лишает силы это полезное волшебство), вы получите доступ к дубликату дескриптора. В некоторых отношениях такая конструкция более переносима, чем передача файла из */dev/fd/*, поскольку она работает везде, где работает Perl, а не только в системах, где есть каталог */dev/fd/*. С другой стороны, такой специальный синтаксис open для доступа к дескрипторам файлов по номеру

---

[1] Можно вызвать stat для каждого дескриптора и сравнить полученные значения (пары device/inode). Но тогда будет уже слишком поздно, потому что повреждения уже нанесены. Все, что можно сделать, – это обнаружить ущерб и произвести аварийный выход (и, может быть, послать письмо системному администратору).

работает только с программами Perl и не работает с программами, написанными на других языках.

Работа с ненадежным кодом

Механизм меченых данных – это лишь защитный барьер перед фальшивыми данными, которые вы должны были бы перехватить сами, но не позаботились об этом и передали их в систему. Он напоминает необязательные предупреждения, которые умеет выдавать Perl – они не всегда свидетельствуют о настоящей проблеме, но в среднем потери от реакции на ложную проблему меньше потерь от отсутствия реакции на настоящую. Что касается меченых данных, потери второго рода еще более заметны, поскольку фальшивые данные не просто дают неправильный ответ, они могут загубить систему и пару предыдущих лет работы. (А может быть, и пару последующих лет, если вы не сделали хороших резервных копий.) Режим проверки меченых данных полезен, когда вы уверены, что написали качественный код, но не очень верите тем, кто снабжает вас данными, и допускаете, что они обманным путем могут попытаться вовлечь вас в нечто крайне нежелательное.

Данные – это одна сторона вопроса. Совсем другая сторона – когда вы не доверяете даже коду, который выполняете. А что если вы загрузите из Сети приложение с вирусом, или бомбу с часовым механизмом, или Троянского коня? Проверка меченых данных при этом будет бесполезна, потому что загруженные данные могут быть прекрасными; не вызывает доверия сам код. Вы ставите себя в такое же положение, как при получении от незнакомого человека неизвестного устройства с запиской, где сказано: «Просто приложите это к голове и потяните за курок». Возможно, вы решите, что это устройство для сушки волос, но вы недолго будете так думать.

В этой области осторожность является синонимом паранойи. Что вам нужно, так это система, позволяющая поместить подозрительный код в карантин. Код продолжит существовать и даже будет выполнять некоторые операции, но он не сможет лазать всюду и делать что заблагорассудится. В Perl такого рода карантин можно осуществить с помощью модуля Safe.

Изменение корневого каталога

Функция chroot в Perl действует точно так же, как системный вызов *chroot*(2). Она изменяет корневой каталог, вследствие чего программа лишается возможности получать доступ к файлам за пределами сегмента файловой системы, выделенного для работы. Однако только суперпользователь root может воспользоваться этой функцией, а это уже проблема безопасности:

```
chroot( '/usr/local/apache/data' );
chdir( '/' );    # теперь в /usr/local/apache/data
```

В действительности это не закрывает доступ к файлам, находящимся за пределами нового корневого каталога. Если до вызова chroot был получен дескриптор каталога, его можно использовать для возврата к оригинальному корневому каталогу даже при том, что вы не сможете использовать имя файла:

```
use v5.14;
use warnings;
```

```
say "Я здесь";

opendir my $rootdh, '/';

chroot( '/Users/Amelia' );
opendir my $dh, '/';    # /Users/Amelia
say for readdir($dh);

chdir( $rootdh );       # Оп-ля, возврат к действительному '/'
opendir my $dh, '.';
say for readdir($dh);
```

Возникновение такой ситуации требует вашего соучастия или соучастия другого человека. Если кто-то может отредактировать программу, добавив код, возвращающий программу в прежний корневой каталог, никакие защитные механизмы Perl не помогут. Используйте любые ухищрения, какие только сможете найти, чтобы исключить возможность запуска программы на Perl с привилегиями суперпользователя.

Защищенные разделы

Модуль Safe позволяет создать некий особый раздел – *sandbox* (*песочницу*), в котором перехватываются все системные операции и тщательно контролируется доступ к пространству имен. Низкоуровневые технические детали этого модуля находятся в процессе непрерывного изменения, поэтому мы предпочли более философский подход.

Ограничение доступа к пространству имен

В самых общих чертах объект Safe напоминает сейф, только идея состоит в том, чтобы нехороших людей запирать внутри сейфа. В мире UNIX существует системный вызов с именем *chroot*(2), который может установить для процесса постоянное ограничение, заставив его выполняться в некотором подкаталоге структуры каталогов, – создать для него, если хотите, маленький персональный ад. Помещенный туда процесс лишен возможности обращаться к файлам, находящимся снаружи, поскольку не имеет способа *назвать* эти файлы.[1]

Объект Safe делает нечто похожее, только ограничивается не подмножеством структуры каталогов файловой системы, а подмножеством структуры пакетов Perl, которая тоже иерархична, как и файловая система.

Можно и по-другому взглянуть на объект Safe – как на специальную комнату с полупрозрачными зеркалами, в которую полиция помещает подозрительных типов. Снаружи в комнату заглянуть можно, но изнутри не видно, что делается снаружи.

При создании объекта Safe можно дать ему имя пакета. Если этого не сделать, новое имя будет выбрано за вас:

```
use Safe;
my $sandbox = Safe->new("Dungeon");
$Dungeon::foo = 1;      # Однако прямой доступ не одобряется.
```

---

1 На некоторых сайтах это делается для выполнения всех сценариев CGI, с применением «закольцованного» монтирования «только для чтения». Такая настройка требует усилий, но если кто-нибудь когда-нибудь сбежит, то обнаружит, что идти ему некуда.

К переменным и функциям пакета можно обращаться извне, если полностью квалифицировать их именем пакета, переданного методу new, по крайней мере в текущей реализации.

Несколько более совместимым с будущими реализациями может оказаться выполнение действий до создания Safe, как показано ниже. Такой код наверняка сохранит работоспособность в будущем и представляет собой удобный способ настройки объекта Safe, который должен начинать работу с большим числом «состояний». (Мы согласны, что $Dungeon::foo не является большим числом состояний.)

```
use Safe;
$Dungeon::master = 'Gary Gygax';    # Все еще прямой доступ, все еще не одобряется.
my $sandbox = Safe->new("Dungeon")
```

Но Safe предоставляет способ доступа к глобальным переменным раздела, даже если вы не знаете имени его пакета. Поэтому для максимальной совместимости в будущем (в ущерб скорости) мы рекомендуем метод reval:

```
use Safe;
my $sandbox = Safe->new();
$sandbox->reval( q($master = 'Gary Gygax') );
```

(В действительности именно этот метод применяется для выполнения подозрительного кода.) Код, передаваемый в безопасный раздел для компиляции и выполнения, считает, что он на самом деле находится в пакете main. То, что внешней среде известно как $Dungeon::master, код внутри рассматривает как $main::master или $::master, или просто $master (если не включена директива use strict). Внутри раздела обращение $Dungeon::master не будет работать, поскольку на самом деле оно будет обращением к $Dungeon::Dungeon::master. Так как объект Safe навязывает собственное представление о main, переменные и подпрограммы в остальной части вашей программы оказываются защищенными.

Чтобы скомпилировать и запустить код внутри раздела, вызовите метод reval («restricted eval»), передав строку кода в аргументе. Как и в любой другой конструкции eval *STRING*, ошибки компиляции и исключительные ситуации времени выполнения в reval не вызовут аварийное завершение вашей программы. Они просто прервут выполнение reval и оставят исключительную ситуацию в переменной $@, поэтому обязательно проверяйте ее после каждого вызова reval.

С учетом произведенной ранее инициализации этот код выведет «master теперь Dave Arneson», но только если вы разрешите вызывать функцию print (см. следующий раздел):

```
$sandbox->permit( qw(print) );
$sandbox->reval(
    q($master = 'Dave Arneson'; print "master теперь $main::master\n")
);
if ($@) {
    die "Невозможно скомпилировать код в разделе: $@";
}
```

Чтобы только скомпилировать код, не выполняя его, заключите строку в объявление подпрограммы:

```
$sandbox->reval(q{
    our $master;
    sub say_master {
```

```
        print "master теперь $main::master\n";
    }
}, 1);
die if $@; # проверка компиляции
```

На этот раз мы передали reval второй аргумент, который, будучи истинным, требует от reval компилировать код с директивой strict. Из самой же строки кода отключить strict нельзя, потому что импорт и отмена импорта относятся к категории операций, которые нельзя обычным образом выполнить в разделе Safe. В Safe запрещено многое – см. следующий раздел.

Если создать в разделе функцию say_master, следующие строки будут действовать примерно одинаково:

```
$sandbox->reval("say_master()");       # Лучший способ.
die if $@;

$sandbox->varglob("say_master")->();   # Вызов через анонимную глобальную.

Dungeon::say_master(),                 # Прямой вызов, очень не одобряется
```

Ограничение доступа к операторам

Другая важная особенность объекта Safe состоит в том, что Perl ограничивает перечень операций, доступных в песочнице (sandbox). (Вы можете позволить своему ребенку взять в песочницу ведерко и лопатку, но на базуку, вероятно, наложите запрет.) Недостаточно защитить оставшуюся часть программы, нужно защитить и оставшуюся часть компьютера.

При компиляции кода в объекте Safe с помощью reval или rdo (ограниченная версия оператора do *FILE*) компилятор сверяется с особым списком управления доступом (он свой у каждого раздела), и проверяет допустимость компиляции каждой отдельной операции. Благодаря этому не нужно проявлять (особое) беспокойство относительно непредвиденных управляющих символов интерпретатора команд, незапланированного открытия файлов, странных утверждений с фрагментами кода в регулярных выражениях и большинства проблем с внешним доступом, по поводу которых обычно волнуются (или должны волноваться) разработчики.

Если потребуется изменить перечень допустимых или запрещенных операций, можно напрямую сообщить разделу, какие операции разрешены или запрещены:

```
use v5.10,

$time = $sandbox->reval( q(time) ), # действует

$sandbox->deny( qw(time) );
$time = $sandbox->reval( q(time) )  # ошибка
```

Можно даже ограничивать доступ к целым группам операций, определяемым в модуле Opcode (табл. 20.1), однако для этого необходимо знать внутреннее устройство *perl*:

```
$sandbox->deny( qw(:base_math) );
my $time = $sandbox->reval( 'log(10)' ); # ошибка
```

Вся хитрость состоит в том, чтобы защитить сам модуль Opcode так, чтобы экспортируемые им теги действительно соответствовали вашим ожиданиям. Если теги

не вызывают доверия, можно указать отдельные операции, имена которых приводятся в описании модуля Opcode. Никогда и никому не доверяйте.

Таблица 20.1. Отдельные теги групп операций из модуля Opcode

Тег	Включает
:base_io	Операции ввода/вывода на основе дескрипторов файлов
:dangerous	Включает массу различных, потенциально опасных операций
:filesys	Ввод и вывод
:load	Загрузка внешних файлов или получение информации о вызывающей программе
:sys_db	Доступ к системной базе данных, такой как */etc/passwd*
:subprocess	Создание порожденных процессов

Модуль Safe не предоставляет полной защиты от атак типа *отказа в обслуживании* (DoS attack), особенно при использовании в менее ограниченных режимах. Атаки типа «отказ в обслуживании» захватывают все имеющиеся в системе ресурсы определенного вида, в результате чего другие процессы не могут получить доступ к важным системным средствам. Примерами таких атак служат переполнение таблицы процессов ядра, захват CPU с помощью жесткого бесконечного цикла, израсходование имеющейся памяти и переполнение файловой системы. Такие проблемы очень тяжело решать, особенно переносимым способом. Атаки типа «отказ в обслуживании» рассматриваются в конце раздела «Код, маскирующийся под данные».

Примеры использования модуля Safe

Представьте, что у вас есть программа CGI, управляющая формой, куда пользователь может ввести произвольное выражение Perl и получить результат его выполнения.[1] Как и любые внешние данные, эта строка будет меченой, поэтому Perl пока не позволит ее выполнить: сначала нужно снять признак мечености с помощью поиска по шаблону. Проблема в том, что невозможно разработать шаблон, способный выявить любую возможную угрозу. А вам не следует просто снимать признак мечености с любых получаемых данных и отправлять их во встроенную функцию eval. (Если вы сделаете это, мы почувствуем соблазн взломать вашу систему и удалить этот сценарий.)

Здесь приходит на помощь reval. Следующий сценарий CGI обрабатывает форму с единственным полем, вычисляет (в скалярном контексте) выражение в строке и выводит результат:

```
#!/usr/bin/perl -lTw
use strict;
use CGI::Carp "fatalsToBrowser";
use CGI qw/:standard escapeHTML/;
use Safe;

print header(-type => "text/html;charset=UTF-8"),
      start_html("Perl Expression Results"),
```

[1] Не надо смеяться. Мы действительно встречали веб-страницы, которые это делают. Без Safe!

```
my $expr = param("EXPR") =~ /^([^;]+)/
            ? $1 # возвращает часть, теперь уже не меченную
            : croak("no valid EXPR field in form");
my $answer = Safe->new->reval($expr),
die if $@;

print p("Result of", tt(escapeHTML($expr)),
              " is", tt(escapeHTML($answer)));
```

Теперь представьте себе зловредного пользователя, который введет строку "print `cat /etc/passwd`" (или что-нибудь похуже). Благодаря окружению с наличием ограничений, которое запрещает использовать символ обратной косой черты, Perl распознает проблему во время компиляции и немедленно вернет управление. В переменной $@ будет строка "quoted execution (``, qx) trapped by operation mask" (выполнение в кавычках, перехваченное маской операций) с дополнительной информацией о том, где возникла эта проблема.

Поскольку мы не указывали иного, наши защищенные разделы до сих использовали установленный по умолчанию набор разрешенных операций. Сейчас не важно, каким образом задаются разрешенные или запрещенные операции. Важно, что это полностью находится под контролем вашей программы. А поскольку в программе можно создать несколько объектов Safe, то допустимо назначать разные степени доверия разным фрагментам кода в зависимости от источника их получения.

Для желающих поэкспериментировать с Safe ниже приводится маленький интерактивный калькулятор на Perl. Он позволяет вводить числовые выражения и немедленно получать их значения. Но он не ограничен только числами. Он больше похож на пример цикла в описании eval из главы 27 – там можно взять все что угодно, вычислить и получить результат. Разница в том, что версия с Safe не выполняет все что угодно. Вы можете запустить этот калькулятор интерактивно в своем терминале, вводить фрагменты кода Perl и просматривать результаты, чтобы понять, какого рода защиту предоставляет Safe.

```
#!/usr/bin/perl -w
# safecalc - демо-программа для проверки Safe
use strict;
use Safe;
my $sandbox = Safe->new();
while (1) {
    print "Input: ";
    my $expr = <STDIN>;
    exit unless defined $expr;
    chomp($expr);
    print "$expr produces ";
    local $SIG{__WARN__} = sub { die @_ };
    my $result = $sandbox->reval($expr, 1);
    if ($@ =~ s/at \(eval \d+\).*//) {
        printf "[%s]: %s", $@ =~
            /trapped by operation mask/
            ? "Security Violation" : "Exception", $@;
    }
    else {
        print "[Normal Result] $result\n";
    }
}
```

Если передать сценарию обычное алгебраическое выражение, он вычислит выражение и вернет результат. Если попытаться сделать что-то другое, например выполнить команду в обратных апострофах или загрузить модуль, вы получите отказ:

```
Input: 2+2
2+2 produces [Normal Result] 4
Input: `ls -l`
`ls -l` produces [Security Violation]: quoted execution (``, qx)'
trapped by operation mask
Input: use LWP::Simple; getprint('http://www.perl.org')
use LWP::Simple; getprint('http://www.perl.org') produces [Security Violation]:
'require' trapped by operation mask
Input: 1/137
1/137 produces [Normal Result] 0.0072992700729927
```

Код, маскирующийся под данные

Разделами Safe можно пользоваться при обработке действительно опасных данных, но это не значит, что защитой можно пренебречь в повседневной работе по дому. Необходимо совершенствовать свое знание окружающей местности и смотреть на вещи с точки зрения субъекта, желающего проникнуть внутрь. Необходимо предпринимать упреждающие меры, например, поддерживать хорошее освещение и подстригать кусты, в которых могут притаиться проблемы.

Perl старается помочь и в этом тоже. Принятая в Perl схема синтаксического анализа и выполнения позволяет избежать ошибок, жертвами которых часто становятся языки сценариев интерпретаторов команд. Язык обладает многими очень мощными возможностями, но они умышленно синтаксически и семантически ограничены таким образом, чтобы все находилось под контролем программиста. За редкими исключениями, Perl интерпретирует каждую лексему только один раз. Если нечто используется как простая переменная с данными, оно не начнет вдруг шарить в вашей файловой системе.

К сожалению, такое может случиться при обращении к интерпретатору команд для запуска других программ, потому что тогда вы будете работать по правилам интерпретатора команд, а не Perl. Впрочем, без интерпретатора легко обойтись – нужно лишь использовать формы функций system, exec или open, открывающей канал, со списком аргументов. Для обратных апострофов нет формы вызова со списком аргументов, защищающей от интерпретатора команд, но их всегда можно эмулировать, как описано выше в разделе «Доступ к командам и файлам с ограниченными привилегиями». (Синтаксически нельзя заставить обратные апострофы принимать список аргументов, но сейчас разрабатывается форма лежащего в их основе оператора readpipe со многими аргументами; на момент написания книги она еще не вполне была готова.)

Когда переменная используется в выражении (в том числе интерполируется в строку в двойных кавычках), Совершенно Исключено, что код Perl, содержащийся в такой переменной, сделает нечто, вами не предусмотренное.[1] В отличие

[1] Хотя, если вы генерируете веб-страницу, можно получить на выходе теги HTML, в том числе код JavaScript, которые сделают нечто, чего удаленный браузер не ожидает.

от интерпретатора команд, Perl не требует охранительных кавычек вокруг переменных, что бы в них ни содержалось.

```
$new = $old;                  # Кавычки не требуются
print "$new items\n";         # $new не навредит вам

$phrase = "$new items\n";     # И здесь тоже.
print $phrase;                # По-прежнему полный порядок
```

В Perl реализован подход «what you see is what you get» (что видишь, то и получаешь). Если вы не видите дополнительного уровня интерполяции, значит, его нет. *Можно* интерполировать в строки произвольные выражения Perl, но только если специально попросить об этом Perl. (И даже в этом случае содержимое подвергается проверке меченых данных, если включен этот режим.)

```
$phrase = "You lost @{[ 1 + int rand(6) ]} hit points\n";
```

Однако интерполяция не является рекурсивной. Нельзя спрятать в строке произвольное выражение:

```
$count = "1 + int rand(6)";           # Некоторый случайный код
$saying = "$count hit points";        # Просто литерал
$saying = "@{[$count]} hit points";   # Тоже литерал
```

Оба присваивания $saying создают "1 + int rand(6) hit points", не интерпретируя интерполированное содержимое $count как код. Чтобы заставить Perl это сделать, нужно явно вызвать eval *STRING*:

```
$code = "1 + int rand(6)";
$die_roll = eval $code;
die if $@;
```

Будь $code меченой, вызов eval *STRING* создал бы исключительную ситуацию. Конечно, почти никогда не требуется вычислять случайный код, но если вдруг придется, то следует подумать об использовании модуля Safe. Возможно, вы уже слышали о таком.

Есть одно место, в котором Perl может иногда обрабатывать данные как код, а именно, когда шаблон в операторах qr//, m// или s/// содержит одно из новых утверждений регулярных выражений, (?{*CODE*}) или (??{*CODE*}). Они не угрожают безопасности, если используются как литералы при поиске по шаблону:

```
$cnt = $n = 0;
while ($data =~ /( \d+ (?{ $n++ }) | \w+ )/gx) {
    $cnt++;
}
print "Got $cnt words, $n of which were digits.\n";
```

Но существующий код, который интерполирует переменные в шаблон, был написан в предположении, что данные – это данные, а не код. Эти новые конструкции могут создать брешь в прежде безопасных программах. Поэтому Perl отказывается интерпретировать шаблон, если интерполируемая строка содержит утверждение с фрагментом кода, и возбуждает исключительную ситуацию. Если такая возможность действительно необходима, ее всегда можно включить директивой use re 'eval' с лексической областью видимости. (Однако по-прежнему нельзя использовать меченые данные для интерполируемого утверждения с фрагментом кода.)

Совершенно иного рода проблемы безопасности, связанные с регулярными выражениями, относятся к отказу в обслуживании. Из-за них программа может закончить работу слишком рано, работать слишком долго, исчерпать всю имеющуюся память, а иногда и потерпеть крах – в зависимости от фазы луны.

При обработке шаблонов, полученных от пользователей, не нужно беспокоиться о возможности интерпретации случайного кода Perl. Однако в механизме регулярных выражений есть свои маленькие компилятор и интерпретатор, а пользовательский шаблон может вызвать у компилятора регулярных выражений изжогу. Если интерполированный шаблон является недопустимым, возникает исключительная ситуация, которая станет фатальной, если не будет обработана. Обрабатывая эту ситуацию, используйте только eval *BLOCK*, а не eval *STRING*, потому что дополнительный уровень интерпретации в последнем может на практике допускать выполнение случайного кода Perl. Вместо этого сделайте нечто вроде следующего:

```
if (not eval { "" =~ /$match/; 1 }) {
    # (Обработать подозрительный шаблон.)
}
else {
    # Мы знаем, чтс шаблон по крайней мере компилируется.
    if ($data =~ /$match/) { ... }
}
```

Более неприятная проблема отказа в обслуживании возникает, когда при правильных данных и правильном шаблоне поиска программа зависает навсегда. Это связано с тем, что для некоторых шаблонов сопоставления время обработки растет экспоненциально и может превысить среднее время наработки на отказ нашей солнечной системы. Если вам особенно повезет, то для такого шаблона с интенсивными вычислениями потребуется еще и экспоненциально растущий объем памяти. В таком случае ваша программа исчерпает всю имеющуюся виртуальную память, утопит остальную часть системы, выведет из себя пользователей и либо закончит существование с обычной ошибкой «Out of memory!», либо оставит после себя громадный файл с дампом памяти, хотя, возможно, он будет не таким большим, как солнечная система.

Как большинство атак типа «отказ в обслуживании», эту проблему нелегко предотвратить. Если платформа поддерживает функцию alarm, можно ограничить время поиска по шаблону. К сожалению, Perl (в настоящее время) не может гарантировать, что обычная обработка сигнала не вызовет сбой программы. (Планируется исправить это в следующих версиях.) Однако вы всегда можете попробовать этим воспользоваться, и даже если сигнал не удастся красиво обработать, по крайней мере, программа не будет работать вечно.

Если ваша система поддерживает ограничение ресурсов, выделяемых каждому процессу, можете установить свои ограничения из интерпретатора команд, прежде чем запускать программу Perl, либо использовать модуль BSD::Resource из CPAN, чтобы сделать это прямо из Perl. Веб-сервер Apache позволяет устанавливать границы времени, памяти и размера файла для запускаемых им сценариев CGI.

Наконец, мы надеемся, что вы дочитываете эту главу, испытывая гнетущее чувство опасности. Помните, что если вы параноик, то это не значит, что за вами никто не гонится. И раз уж на то пошло, вы можете получить от этого удовольствие.

21

Распространенные приемы программирования

Любой программист на Perl с удовольствием даст вам массу советов, как программировать. Мы не исключение (если вдруг вы еще не поняли). В данной главе мы не станем рассказывать о специфических возможностях Perl, а зайдем с другого конца и используем более вольный подход для описания идиом Perl. Надеемся, что связав, казалось бы, несвязанные вещи, вы сможете в большей мере проникнуться ощущением того, что означает «думать на Perl». В конце концов, когда вы программируете, вы не создаете сначала несколько выражений, затем несколько подпрограмм и в конце несколько объектов. В какой-то мере приходится заниматься всем этим одновременно. Так и в этой главе – немного обо всем сразу.

Однако элементарная организация здесь все-таки есть, и проявится она в том, что, начав с отрицательных примеров, мы будем продвигаться в направлении положительных. Не знаем, станет ли от этого легче вам, но нам – станет. Кроме того, на протяжении всей своей карьеры программиста вы будете учиться тому, что не следует делать, прежде чем научитесь делать то, что следует, так что начинайте привыкать заранее.

Обычные промахи новичков

Самая большая ошибка – забыть включить вывод предупреждений директивой `use warnings`, идентифицирующей многие ошибки. Вторая по масштабам ошибка – забыть вставить `use strict`, где она уместна. Эти две прагмы могут уберечь от долгих часов бесплодных раздумий, когда ваша программа начнет разрастаться в объеме. (А она начнет.) Еще одной оплошностью будет забыть проконсультироваться с *perlfaq*. Допустим, вы хотите узнать, есть ли в Perl функция `round`. Сначала можно попытаться поискать в распространенных вопросах – при помощи *perldoc*:

```
% perldoc -q round
```

Помимо этих «метапромахов» существует ряд ловушек программирования. В некоторые из них попадает почти каждый, а в другие можно угодить, только придя

из другой культуры, где многое делается иначе. Ошибки эти сгруппированы в следующих разделах.

Всеобщие ошибки

- Запятая после дескриптора файла в команде print. Хотя кажется совершенно нормальным и приличным сказать:

  ```
  print STDOUT, "goodbye", $adj, "world!\n"; # НЕВЕРНО
  ```

  это, тем не менее, некорректно из-за этой первой запятой. Вместо этого здесь нужен синтаксис косвенного объекта:

  ```
  print STDOUT "goodbye", $adj, "world!\n", # ok
  ```

  Этот синтаксис позволяет сказать:

  ```
  print $filehandle "goodbye", $adj, "world!\n";
  ```

  где $filehandle – скаляр, содержащий дескриптор файла на этапе выполнения. Это не то же самое, что:

  ```
  print $notafilehandle, "goodbye", $adj, "world!\n";
  ```

  где $notafilehandle интерпретируется как часть списка выводимых данных. В данном случае вы увидите в терминале нечто подобное: GLOB(0xDEADBEEF), потому что будет выполнен вывод в стандартный поток вывода ссылки на дескриптор файла в строковой форме.

  См. «косвенный объект» в глоссарии.

- Применение == вместо eq и != вместо ne. Операторы == и != используются для сравнения *чисел*. Другие два оператора выполняют проверки *строк*. Строки "123" и "123.00" равны как числа, но не равны как строки. Кроме того, любая нечисловая строка численно равна нулю, и некоторые из них, такие как "123xyz", вы, вероятно, не собирались использовать в числовом контексте. За исключением ситуаций, когда вы работаете с числами, почти всегда следует использовать операторы сравнения строк. Прагма warnings сообщит о попытках использовать эти операторы в нечисловом контексте.

- Отсутствие замыкающей точки с запятой. Каждая команда в Perl завершается точкой с запятой или концом блока. Символы перевода строки не служат окончанием команды, как в *awk*, Python или FORTRAN. Помните, что Perl похож на C.

  Инструкция, содержащая внедренный документ, особенно чувствительна к потере точки с запятой. Она должна выглядеть так:

  ```
  print <<'FINIS';
  A foolish consistency is the hobgoblin of little minds,
  adored by little statesmen and philosophers and divines.
                                          --Ralph Waldo Emerson
  FINIS
  ```

- Отсутствие фигурных скобок при *BLOCK*. Простые инструкции не являются блоками. При создании управляющей структуры, такой как while или if, состоящей из одного или более блоков, необходимо заключать каждый блок в фигурные скобки. Помните, что Perl – это *не* C.

- **Забывают сохранять переменные $1, $2 и т.д., при переходе к новому регулярному выражению. Помните, что каждая успешная операция** m/atch/ **или** s/ubsti/tution/ **изменяет (или чистит, или портит) переменные $1, $2, ... Один из способов сразу сохранить их – выполнить поиск в списочном контексте:**

```
my ($one, $two) = /(\w+) (\w+)/;
```

- **Многие не понимают, что** local **изменяет значение переменной, которое видят другие подпрограммы, вызываемые в пределах области видимости локальной переменной. Легко забыть, что** local **является инструкцией этапа выполнения, которая создает динамическую область видимости, поскольку для нее нет эквивалента в языках типа С. См. раздел «Объявления с областью видимости» в главе 4. В любом случае обычно требуется** my **вместо** local**.**
- **Потеря парности фигурных скобок. Хорошие текстовые редакторы помогают находить пары. Обзаведитесь таким. (Или такими.) Немалую помощь окажет следование определенному стилю расположения скобок, благодаря чему вы будете знать – где ожидать их, даже если другие готовы драться, доказывая, что это неправильно. Такие инструменты, как** Perl::Tidy, **помогут вам отформатировать ваш код.**
- **Применение инструкций управления циклами в** do {} while. **Хотя фигурные скобки в этой конструкции выглядят подозрительно похожими на части блока цикла, они таковыми не являются.**
- **Использование** $foo[1], **когда имеется в виду** $foo[0]. **Нумерация элементов в массивах Perl по умолчанию всегда начинается с нуля. В прежние времена Perl пытался быть гибким, позволяя начинать нумерацию с любого числа с помощью специальной переменной** $[, **но начиная с версии v5.12 этот прием считается устаревшим.**
- **Использование** @foo[0], **когда имеется в виду** $foo[0]. **Ссылка** @foo[0] **является *срезом* массива, т.е. массивом, состоящим из единственного элемента** $foo[0]. **Иногда никакой разницы нет, как в:**

```
print "the answer is @foo[0]\n";
```

но она очень велика в следующем случае:

```
@foo[0] = <STDIN>;
```

когда будет «проглочено» все, оставшееся в STDIN, **переменной** $foo[0] **будет присвоена первая строка, а все остальное будет отброшено. Едва ли это входило в ваши намерения. Приучите себя думать, что** $ **соответствует единственному значению, а** @ **– списку значений, и все у вас будет в порядке.**

- **Потеря круглых скобок в списочном операторе, таком как** my, **что делает одну переменную лексической, а другую – глобальной:**

```
my $x, $y = (4, 8);    # НЕВЕРНО
my ($x, $y) = (4, 8); # ok
```

- **Забывают выбрать правильный дескриптор файла, прежде чем установить** $^, $~ **или** $|. **Эти переменные зависят от дескриптора файла, выбранного в данный момент посредством** select(*FILEHANDLE*). **Первоначальным выбранным дескриптором файла является** STDOUT. **Вместо этого следует использовать методы работы с дескрипторами файлов из модуля** IO::Handle. **См. главу 25.**

- Забывают определить кодировку для каждого текстового потока ввода/вывода. Не существует такого универсального понятия, как «текстовый файл». Ключ *-C* командной строки, переменная среды PERL_UNICODE и прагма open позволяют устанавливать кодировку, которая будет использоваться неявно, а функции binmode и open позволяют указывать кодировку явно. Если вы забудете установить кодировку явно или неявно, вы не получите текстовые данные. Кодировку нельзя угадать, она должна быть указана.

Часто игнорируемые советы

Практикующие программисты на Perl должны учесть следующее:

- Многие операции по-разному ведут себя в списочном и скалярном контекстах. Например:

```
($x) = (4, 5, 6);   # Списочный контекст; $x устанавливается в 4
 $x  = (4, 5, 6);   # Скалярный контекст: $x устанавливается в 6

@a = (4, 5, 6);
$x = @a;            # Скалярный контекст: $x устанавливается в 3 (длина массива)
```

- Желательно избегать голых (bare) слов, особенно если они написаны целиком в нижнем регистре. Нельзя, лишь посмотрев на слово, сказать, функция это или просто строка. Используя кавычки для строк и круглые скобки для аргументов функций, вы никогда их не спутаете. На практике директива use strict в начале программы делает голые слова ошибкой этапа компиляции, что, вероятно, правильно.
- Невозможно по внешнему виду определить, какие встроенные функции являются унарными операторами (как chop и chdir), какие – списочными (как print и unlink), а какие не имеют аргументов (как time). Это вы сможете узнать из главы 27. Как всегда, если вы не уверены, используйте круглые скобки – даже если вы не уверены в том, что вы не уверены. Заметьте также, что определенные пользователем подпрограммы по умолчанию являются списочными операторами, но они могут быть объявлены как унарные операторы с помощью прототипа ($) или как не имеющие аргументов с помощью прототипа ().
- Трудно запомнить, что некоторые функции по умолчанию принимают в качестве аргументов $_, @ARGV или что-либо еще, а другие – нет. Потратьте время, чтобы запомнить, «кто есть кто», либо избегайте использования аргументов по умолчанию.
- <FH> не является дескриптором файла; это оператор угловых скобок, осуществляющий операцию чтения строк из дескриптора. Эта путаница обычно проявляется, когда пытаются выполнять print в этот оператор:

```
print <FH> "hi"  # НЕВЕРНО, опустите угловые скобки
```

- Помните, что данные, которые читает оператор угловых скобок, присваиваются переменной $_, только когда чтение файла является единственным условием в цикле while:

```
while (<FH>) { } # Данные присваиваются $_.
<FH>;            # Данные читаются и отбрасываются!
```

- Не применяйте =, когда требуется =~; это разные конструкции:

```
$x =  /foo/;  # Ищет "foo" в $_, помещает результат в $x
$x =~ /foo/;  # Ищет "foo" в $x, отбрасывает результат
```

- **Используйте /r в операциях подстановки для возврата результата.**

```
@new = map { s/old/new/r } @old;
```

- **Используйте my для локальных переменных, если можно обойтись этим. Ключевое слово local просто назначает глобальной переменной временное значение, что оставляет вероятность возникновения непредвиденных побочных эффектов динамических областей видимости.**
- **Избегайте применения local с экспортированными переменными модуля. Если вы локализуете экспортированную переменную, ее экспортированное значение не изменится. Локальное имя станет псевдонимом нового значения, но внешнее имя остается псевдонимом оригинала.**

Ловушки C

Мозговитые программисты на C должны учесть следующее:

- **В блоках if и while фигурные скобки обязательны.**
- **Вы должны использовать elsif, а не «else if» или «elif». Следующий синтаксис:**

```
if (expression) {
    block;
}
else if (another_expression) { # НЕВЕРНО
    another_block;
}
```

является недопустимым. Часть else всегда является блоком, а голый if не является блоком. Не ждите от Perl точного сходства с C. Вот что вам нужно:

```
if (expression) {
    block;
}
elsif (another_expression) {
    another_block;
}
```

Обратите также внимание, что «elif» – это перевернутый «file». Только разработчикам на Algol могло понадобиться ключевое слово, которое совпадает с другим словом, записанным наоборот.

- **Ключевые слова break и continue из C превратились в Perl в last и next соответственно. В отличие от C, они *не* работают в конструкции do {} while.**
- **В Perl долгое время отсутствовал эквивалент инструкции switch из C. В Perl версии v5.10 появилась «инструкция switch на стероидах» с причудливым именем given-when (потому что это действительно весьма причудливая конструкция). См. главу 4. Кроме того, аналог инструкции switch легко построить из имеющихся конструкций; см. «Голые блоки» и «Инструкция given» в главе 4.)**
- **Имена переменных в Perl начинаются с $, @ или %.**
- **Комментарии начинаются с #, а не /*. Для оформления многострочных комментариев используйте синтаксис определения встроенного документа.**

- Нельзя получить адрес чего-либо, хотя в Perl имеется аналогичный оператор (обратная косая черта), создающий ссылку. При преобразовании ссылки в строку можно получить некое подобие адреса, но вы все равно не сможете воспользоваться им.
- Имя ARGV должно записываться в верхнем регистре. $ARGV[0] – это argv[1] в C, а argv[0] из C хранится в $0. См. главу 25.
- Системные вызовы, такие как link, unlink и rename, возвращают в случае успеха истину, а не 0.
- Обработчики сигналов в %SIG работают с именами, а не номерами сигналов.

Ловушки интерпретатора команд

Сообразительные разработчики сценариев для интерпретатора команд (shell) должны принять во внимание следующее:

- Переменные в левой части присваивания имеют префиксы $, @ или %, как и в правой. Присваивание в стиле интерпретатора команд:

  ```
  camel="dromedary", # НЕВЕРНО
  ```

  будет воспринято не так, как вам бы хотелось. Правильно так:

  ```
  $camel="dromedary"; # ok
  ```

- Переменная цикла в foreach тоже требует $. Тогда как в *csh* допустимо:

  ```
  foreach hump (one two)
      stuff_it $hump
  end
  ```

  в Perl это записывается так:

  ```
  foreach $hump ("one", "two") {
      stuff_it($nump);
  }
  ```

- Оператор обратного апострофа выполняет интерполяцию переменных, не обращая внимания на присутствие одинарных кавычек в команде.
- Оператор обратного апострофа не производит трансляцию возвращаемого значения. В Perl необходимо отбросить символ перевода строки явно, например так:

  ```
  chomp($thishost = `hostname`);
  ```

- Интерпретаторы команд (особенно *csh*) выполняют несколько проходов подстановки в каждой командной строке. Perl осуществляет интерполяцию только в определенных конструкциях, таких как двойные кавычки, обратные апострофы, угловые скобки и шаблоны для поиска.
- Интерпретаторы команд обычно выполняют сценарии по частям. Perl целиком компилирует программу перед ее выполнением (за исключением блоков BEGIN, которые выполняются до того, как завершится компиляция).
- Доступ к аргументам программы осуществляется через @ARGV, а не $1, $2 и т.д.
- Доступ к среде в виде отдельных скалярных переменных не предоставляется автоматически. Если вам это требуется, используйте стандартный модуль Env.

Ловушки Python

Оба языка, Perl и Python, являются динамическими, имеют общие корни и появились с разницей в пять лет (1987 и 1991). Более того, Perl 4 даже заимствовал объектную систему из Python для Perl 5. Эти два языка имеют массу сходств, если не обращать внимания на синтаксис, но многое в них реализовано совершенно по-разному.

- Python и Perl иногда используют разные слова для описания одних и тех же понятий, а иногда одинаковые – для разных понятий; см. табл. 21.1.

Таблица 21.1. Соответствие некоторых понятий в Python и Perl

Python	Perl
Кортеж	Список
Список	Массив
Словарь	Хеш

- Переменные в Perl начинаются с `$`, `@` или `%`. Использование разыменовывающих символов, например `$str`, позволяет языку Perl отделять существительные от глаголов, поэтому вы никогда по ошибке не переопределите встроенные имена, что допускает Python, если попытаетесь использовать имя `str` для своих нужд. Perl позволяет переопределять встроенные имена, но никогда – случайно, как в Python.
- Не забывайте использовать `use warnings`, чтобы позволить Perl выводить предупреждения в ситуациях, которые в Python порождают исключения. В противном случае в Perl вы узнаете о многих интересных вещах, только если специально начнете проверять их; т.е. забыв об этой прагме, вы никогда о них не узнаете. См. также описание `warnings` и `autodie` в главе 27.
- Многие встроенные функции принимают аргументы по умолчанию или определяют действия по умолчанию для наиболее типичных случаев. См. главу 27.
- Методы в языке Python принимают явные списки параметров. В Perl принято распаковывать аргументы внутри функции, что обеспечивает большую гибкость в количестве аргументов и порядке их следования. Мы считаем это особенностью, но если вам покажется, что приходится слишком много шаблонного кода для распаковки аргументов функций, посмотрите в сторону модуля `Method::Signatures` из CPAN.
- Шаблоны регулярных выражений Perl распознает и компилирует на этапе компиляции программы.
- Конструкция `\N{NAME}` в Perl позволяет определять сокращения, псевдонимы и собственные имена (которые могут быть разными в разных лексических областях видимости), а конструкция `\N{NAME}` в Python – нет.
- Символы в Perl – это абстрактные коды Юникода, а не низкоуровневые коды, как в Python.
- Механизм сопоставления с шаблоном в Perl руководствуется правилами Юникода при сопоставлении без учета регистра символов, а в Python используются правила ASCII, т.е. например, все три сигмы греческого алфавита при сопоставлении без учета регистра в Perl будут признаны совпадающими.

- Функции отображения регистра символов в Perl, такие как uc и lc, следуют правилам Юникода, поэтому они воздействуют на все символы с изменяемым регистром, а не только на буквы.
- Perl распознает (потенциально вложенные) лексические области видимости и поэтому обеспечивает возможность создания полноценных лексических замыканий. В Python этого нет, так что полные лексические замыкания не поддерживаются.
- В Perl обеспечивается полная поддержка приведения регистра символов Юникода, поэтому изменение регистра символов в строке может приводить к ее удлинению. В Python используются лишь простейшие правила приведения регистра символов Юникода (когда они вообще используются), что не позволяет получать столь же хорошие результаты при работе со строками.
- Любые подпрограммы, возвращающие связанные (освященные) ссылки, в Perl считаются конструкторами. Для конструкторов не предусмотрены какие-то особые правила именования.
- Методы в Perl – это обычные методы, и они всегда в качестве бонуса получают своего инвоканта в первом аргументе. Язык Perl сам по себе не предусматривает различий между методами объектов, методами классов или статическими методами, как в Python.
- Объектная ориентированность в Perl является необязательной дополнительной возможностью. Perl не распространяет объектную ориентированность на встроенные типы, если не попросить его об этом явно, – не все в этом языке имеет методы. Однако вам может понравиться прагма autobox.
- В Perl вы вызываете функцию с аргументами:

  ```
  my $string = join("|", qw(Python Perl Ruby) );
  ```

  В Python, вероятно, придется интерпретировать первый аргумент как объект и вызывать его метод:

  ```
  new = "|".join(["Python", "Perl", "Ruby"])
  ```

- В Perl поиск по шаблону не привязывается к какой-то конкретной позиции в тексте, если явно не определить якорные метасимволы в шаблоне. В Python метод re.search() действует аналогично, но метод re.match() выполняет поиск только с начала строки.
- Строки в Perl не являются массивами символов, поэтому к ним неприменимы операции над массивами. Со строками, естественно, используются только строковые операции.
- За исключением самой обратной косой черты или разделителя с обратной косой чертой, Perl никогда не интерпретирует экранированные последовательности с обратной косой чертой в строках, заключенных в одиночные кавычки, как это делает Python. Строки в одиночных кавычках в Perl, такие как '\t', больше напоминают «сырые» строки в Python, такие как r'\t'.
- Обратные апострофы в Perl используются для заключения и выполнения литералов произвольных системных команд и возвращают их вывод, например: $file = `cat foo.c`.
- В Perl нет необходимости предварительно выделять память, как в Python, потому что массивы и другие структуры данных изменяются в размерах по мере

необходимости, иногда в результате *самооживления* или *автовивификации* (*autovivification*). В Python требуется явно увеличивать размеры списков и явно размещать новые списки и словари.

- В случае ошибок при выполнении обычных операций, таких как открытие файла, Python часто возбуждает исключения, тогда как Perl возвращает специальные значения, обычно undef. Это означает, что, не проверив возвращаемое значение, вы пропустите ошибку. Чтобы заставить системные вызовы генерировать исключения, используйте прагму autodie.
- В случае ошибки преобразования строки в число или при попытке интерпретировать undef как определенное значение, Perl не возбуждает исключение по умолчанию. Такое поведение можно изменить объявлением:

```
use warnings FATAL => q(numeric uninitialized);
```

- Списки в Perl никогда не вкладываются друг в друга, даже если указать дополнительные скобки. Используйте квадратные скобки, чтобы создавать вложенные массивы (массивы ссылок).
- Оператор диапазона в Perl включает в диапазон обе границы, т.е. 0..9 включает 0 и 9.
- Интерактивной оболочкой языка Perl является его отладчик (глава 17), но существует еще одна замечательная интерактивная оболочка, Devel::REPL. Вызов Perl без аргументов не запускает интерактивный цикл чтение-вычисление-вывод (read-eval-print), как в Python. Для этого следует использовать команду perl -de0.

Ловушки Ruby

Мацумото Юкихиро (Matz), создатель Ruby, многое заимствовал из Perl (и мы считаем, что он правильно выбрал стартовую позицию). Фактически он поселил вместе Perl и Smalltalk и позволил им дать потомство.

- В Perl нет аналога команды *irb* (интерактивная Ruby-оболочка). См. раздел о Python.
- В Perl есть просто числа. Он не беспокоится о наличии или отсутствии в них дробной части.
- В Perl нет необходимости заключать переменные в {} (#{} в Ruby) для их интерполяции, если только не требуется устранить неоднозначность:

```
"My favorite language is $lang"
```

- Интерполируемые строки в Perl необязательно заключать в двойные кавычки: можно использовать оператор qq с произвольными разделителями. Аналогично, неинтерполируемые строки необязательно заключать в одиночные кавычки: можно использовать оператор q с произвольными разделителями.

```
q/That's all, folks/
q(No interpolation for $100)
qq(Interpolation for $animal)
```

- Инструкции в Perl должны разделяться точкой с запятой (;), даже если находятся в разных строках. Последнюю инструкцию в блоке не требуется завершать точкой с запятой.

- **Регистр символов в именах переменных не несет дополнительной смысловой нагрузки для *perl*.**
- **Разыменовывающие символы не определяют область видимости переменной или ее тип. Символ** `$` **в Perl – это единственный элемент, например:** `$scalar`, `$array[0]` **или** `$hash{$key}`.
- **Сравнение строк в Perl выполняется операторами** `lt`, `le`, `eq`, `ne`, `ge` **и** `gt`.
- **Никаких волшебных блоков, но взгляните на** `PerlX::MethodCallWithBlock`.
- **Функции в Perl определяются на этапе компиляции. То есть фрагмент**

```
use v5.10.
sub foo { say "Camelia" }
foo();
sub foo { say "Amelia" };
foo();
```

  дважды выведет слово `Amelia`**, потому что к моменту начала выполнения программы будет действовать второе определение функции. Это также означает, что вызов подпрограммы может находиться в файле выше ее определения.**
- **В Perl отсутствуют переменные класса, но многие пытаются имитировать их с помощью лексических переменных.**
- **Оператор диапазона в Perl возвращает список, но взгляните на расширение** `PerlX::Range`.
- **Модификатор** `/s` **шаблонов обеспечивает соответствие точки (.) символу перевода строки, тогда как в Ruby ту же роль играет модификатор** `/m`**. Модификатор** `/m` **в Perl обеспечивает соответствие якорных символов** `^` **и** `$` **с началом и концом логических строк.**
- **В Perl используются плоские списки.**
- **Оператор => в Perl может находиться практически везде, где допускается запятая, поэтому в программах на Perl часто можно видеть, как оператор стрелки => используется, чтобы указать направление:**

```
rename $old => $new;
```

- **В Perl значения** `0`, `"0"`, `""`, `()` **и** `undef` **означают «ложь» в логическом контексте. В базовом языке Perl отсутствуют специальные логические значения, однако обратите внимание на модуль** `boolean`.
- **В Perl часто вместо** `nil` **используется** `undef`.
- **Иногда Perl требует особой внимательности, потому что некоторые символы имеют в нем особое значение. Например, вопросительный знак ?, следующий за именем переменной, не является частью этого имени:**

```
my $flag = $foo? 0 :1,
```

Ловушки Java

- **В Perl нет метода** `main` **или ему подобного:**

```
public static void main(String[ ] argv) throws IOException
```

- Perl выделяет память автоматически, по мере роста массивов и хешей. Автооживление означает, что память будет выделена в момент присваивания, если она не была выделена раньше.
- Perl вынуждает объявлять переменные заранее, только если используется прагма `use strict`.
- Perl различает значения и ссылки на значения, т.е. вам (как правило) необходимо явно разыменовывать ссылки.
- В Perl функции не обязаны быть методами.
- Строковые и числовые литералы в Perl обычно объектами не являются, но могут ими быть.
- Программисты на Java, определяющие структуры данных как классы, могут удивиться, обнаружив, что в Perl они создаются на основе простых объявлений, смешивающих анонимные хеши и массивы. См. главу 9.
- Данные экземпляров в Perl являются (обычно) простыми значениями в хеше, используемом в качестве объекта, где имя поля в хеше соответствует имени элемента данных экземпляра в Java.
- Закрытые данные в Perl не являются обязательными.
- Функция, которая в конечном итоге будет вызвана при обращении к методу, неизвестна до момента выполнения, так что может быть вызван любой объект или класс с методом, имеющим это имя. Значение имеет только интерфейс.
- Perl поддерживает перегрузку операторов.
- Perl не поддерживает перегрузку функций по сигнатуре. Обратите внимание на модуль `Class::Multimethod` в CPAN.
- Perl допускает множественное наследование, хотя этот механизм больше напоминает реализацию множества интерфейсов в Java, поскольку классы в Perl наследуют только методы, но не данные.
- Значение типа `char` в Java – это не абстрактный код Юникода; это часть кода символа в кодировке UTF-16. То есть полный код занимает два значения `char` в Java, и требует применять специальные приемы программирования при обработке символов за границами основной многоязычной таблицы (Basic Multilingual Plane) в Java. В Perl, напротив, символ является абстрактным кодом Юникода, фактическая реализация которого преднамеренно скрыта от программиста. Программный код на Perl автоматически работает со всем диапазоном символов Юникода, а также за его пределами.
- В отличие от Java, строковые литералы в Perl могут включать символы перевода строки буквально. Однако внедренные документы все же лучше подходят для этой цели.
- Обычно функции сообщают об ошибке, возвращая `undef`, а не возбуждая исключение. Если вам больше по душе другой путь, используйте прагму `autodie`.
- В Perl не используются именованные параметры; аргументы внутри функций доступны в виде массива `@_`. Однако, обычно им сразу же присваиваются имена. Обратите внимание на модуль `Methods::Signatures` из CPAN, если предпочитаете более формальный способ объявления имен параметров.
- Такие понятия, как прототипы функций Perl, вообще отсутствуют в Java.

- Perl поддерживает передачу параметров по именам, позволяя опускать необязательные аргументы и передавать аргументы в произвольном порядке.
- Система сборки мусора Perl основана на подсчете ссылок, поэтому есть возможность написать деструктор, автоматически освобождающий ресурсы, такие как открытые дескрипторы файлов, соединения с базами данных, блокировки файлов и другие.
- Регулярные выражения Perl не требуют использования дополнительных символов обратной косой черты.
- Литералы регулярных выражений Perl компилирует, проверяя их синтаксис на этапе компиляции, и сохраняет в скомпилированном виде для большей эффективности.
- При сопоставлении с шаблоном Perl не подразумевает наличие неявных якорных символов в шаблоне, как это делает метод `match` в Java. Принцип действия сопоставления с шаблоном в Perl больше напоминает метод `find` в Java.
- Шаблон в Perl может иметь несколько сохраняющих групп с одинаковым именем.
- Шаблоны в Perl могут быть рекурсивными.
- Чтобы задействовать механизм свертки регистра Юникода для сопоставления без учета регистра символов, в шаблонах Java требуется использовать специальный параметр, а в Perl шаблоны используют этот механизм по умолчанию. Кроме того, Perl поддерживает свертку регистра в полном объеме, а в Java используется упрощенный подход.
- В шаблонах Java традиционные классы символов, такие как `\w` и `\s`, по умолчанию охватывают только символы ASCII, а для распространения их действия на символы Юникода требуется передавать специальный параметр. Шаблоны в Perl поддерживают Юникод по умолчанию, поэтому специальный параметр необходимо передавать для сужения области действия классов символов до набора ASCII.
- Механизм JNI в Java соответствует механизму XS в Perl, по крайней мере по духу. Модули в Perl часто включают скомпилированные компоненты на C/C++, а в Java это редкость.
- Не нужно стремиться все переписать на Perl; Perl позволяет легко и просто вызывать системные программы с помощью обратных апострофов, `system` и варианта `open` для открытия каналов.

Эффективность

Часто работа программиста состоит лишь в том, чтобы заставить программу правильно работать, но не исключено, что вы захотите добиться большего от своей программы. Богатый арсенал операторов, типов данных и управляющих конструкций не всегда позволяет интуитивно оценить возможности оптимизации по скорости или памяти. Разработчикам Perl пришлось пойти на многочисленные компромиссы, и такие решения погребены в глубинах кода. В целом, чем короче и проще ваш код, тем он быстрее выполняется, но встречаются исключения. В данном разделе мы попытаемся помочь вам заставить свои программы работать капельку лучше.

Если программа должна работать намного быстрее, поэкспериментируйте с компилятором Perl, описанным в главе 16, или перепишите внутренний цикл как расширение на C (об этом не рассказывается в данной книге). Но, прежде чем что-либо предпринимать, выполните профилирование программы (как описывается в главе 17), чтобы выявить места, где проще всего было бы выполнить необходимые оптимизации.

Обратите внимание, что оптимизация по времени может иногда обернуться нерациональным расходованием памяти или снижением эффективности программирования (на что указывают противоречивые советы ниже). Так уж это устроено. Будь программирование простым делом, для него не потребовалось бы такое сложное создание, как человек, не так ли?

Эффективность по времени

- Используйте хеши вместо линейного поиска. Например, вместо поиска по @keywords, чтобы узнать, является ли $_ ключевым словом, создайте следующий хеш:

  ```
  my %keywords;
  for (@keywords) {
      $keywords{$_}++
  }
  ```

  После этого можно быстро узнать, содержит ли $_ ключевое слово, сравнив $keyword{$_} с нулем.

- Избегайте употребления индексов, когда можно использовать оператор foreach или список. Применение индексов требует дополнительных операций, а так же, если индексы (в результате вычислений) окажутся действительными числами, это потребует дополнительного преобразования их обратно в целые числа. Часто можно отыскать более эффективный способ. Обдумайте возможность применения foreach, shift и splice. Попробуйте сказать use integer.

- Избегайте оператора goto. Он производит сканирование кода от текущего адреса в поисках заданной метки.

- Избегайте printf, если можно обойтись print.

- Избегайте переменной $& и ее подруг $` и $'. Их появление в программе приводит к тому, что при любом успешном поиске строка, в которой производится поиск, сохраняется для возможного использования в будущем. (Однако, если они появились в коде один раз, дальнейшее их использование хуже уже не сделает.) В Perl v5.10 появились отдельные переменные для соответствий, создаваемые модификатором /p (глава 5), которые не заставляют выбирать между эффективностью и удобством.

- Избегайте применения eval к строке. eval строки (к eval *BLOCK* это не относится) требует перекомпиляции при каждом вызове. Анализатор Perl работает довольно быстро, но это ни о чем не говорит. Почти всегда, чтобы решить задачу, есть способ получше. В частности, любой код, использующий eval только для создания имен переменных, является устаревшим, поскольку то же самое можно сделать непосредственно с помощью символических ссылок:

```
no strict "refs";
$name = "variable";
$$name = 7;            # устанавливает $variable равной 7
```

Не то чтобы мы рекомендуем такое решение, но, если вы не найдете другого пути, это решение привлекательнее, чем eval строки.

- Оператор выбора альтернатив часто работает быстрее, чем соответствующее регулярное выражение, поэтому

```
print if /one-hump/ || /two/;
```

наверняка будет выполняться быстрее, чем

```
print if /one-hump|two/;
```

по крайней мере, для некоторых значений one-hump и two. Последнее связано с тем, что оптимизатор любит поднимать некоторые простые операции поиска в верхние части синтаксического дерева и осуществлять очень быстрый поиск по алгоритму Бойера-Мура (Boyer-Moore). Сложный шаблон может помешать этому.

- Используйте next if для упреждающего исключения «обычных» ситуаций. Оптимизатору это нравится, как и простые регулярные выражения. Это также позволяет избежать лишней работы. Обычно можно выкинуть строки комментариев и пустые строки, прежде чем выполнить split или chop:

```
while (<>) {
    next if /^#/;
    next if /^$/;
    chop;
    @piggies = split(/,/);
    ...
}
```

- Избегайте регулярных выражений с большим количеством квантификаторов и большими числами *{MIN, MAX}* для выражений в круглых скобках. Такие шаблоны могут привести к экспоненциальному замедлению поиска с возвратом, если «квантифицированные подшаблоны» не будут найдены при первом «проходе». Можно также использовать конструкцию (?>...), чтобы потребовать полного соответствия шаблона либо признания неудачи без осуществления поиска с возвратом.
- Старайтесь максимально увеличить длину всех обязательных литеральных строк в регулярных выражениях. Интуиция подсказывает обратное, но длинные шаблоны часто отыскиваются быстрее, чем короткие, потому что постоянные строки оптимизатор передает алгоритму Бойера-Мура, который выигрывает от длинных строк. Скомпилируйте шаблон с ключом отладки -Dr и посмотрите, какую строку литералов Dr. Perl считает самой длинной.
- Избегайте дорогостоящих вызовов подпрограмм в глубоких циклах. С вызовом подпрограмм связаны накладные расходы, особенно при передаче длинных списков параметров и возврате длинных значений. В порядке возрастания степени отчаяния попробуйте передавать значения по ссылке или как глобальные переменные с динамической областью видимости, встраивать (inline) подпрограммы или переписать весь цикл на C. (Лучше, если вы сможете вообще избавиться от подпрограмм, применив более умный алгоритм.)

- **Избегайте повторных вызовов одной и той же подпрограммы, если известно, что каждый раз она будет возвращать одно и то же значение. В этом вам помогут такие модули, как** Memoize, **или сконструируйте собственный кэш для хранения известных значений (которые не будут изменяться).**
- **Избегайте вызова** getc **кроме как для ввода одиночных символов с терминала. На самом деле лучше и для этого ее не использовать. Следует предпочесть** sysread.
- **Избегайте частого применения** substr **с длинными строками, особенно если строка содержит текст в кодировке UTF-8. Эту функцию лучше применять к началу строки, а в некоторых задачах можно сделать так, чтобы** substr **работала с началом, «зажевывая» строку по ходу дела с помощью** substr **с четырьмя аргументами и заменяя захваченную часть на** "":

```
while ($buffer) {
    process(substr($buffer, 0, 10, ""));
}
```

- **Используйте** pack **и** unpack **вместо многократных вызовов** substr.
- **Применяйте** substr **в качестве левого значения вместо конкатенации подстрок. Например, для замены символов** $foo **с четвертого по седьмой содержимым переменной** $bar **не делайте так:**

```
$foo = substr($foo,0,3) . $bar . substr($foo,7);
```

Вместо этого просто укажите заменяемую часть строки и выполните присваивание ей:

```
substr($foo, 3, 4) = $bar;
```

Но помните, что если $foo **является очень длинной строкой, а** $bar **не соответствует в точности размеру «дырки», это может привести к большому объему копирования. Perl попытается минимизировать его, выбрав копирование с начала или с конца, но ничего особенно полезного не сможет сделать, если** substr **находится в середине.**

- **Используйте** s/// **вместо конкатенации подстрок. Особенно в случаях, когда можно заменить одну константу другой такого же размера. В результате получится подстановка на месте.**
- **Применяйте модификаторы команд и эквивалентные операторы** and **и** or **вместо полноразмерных условных операторов. Модификаторы команд (такие как** $ring = 0 unless $engaged**) и логические операторы позволяют избежать расходов, связанных с входом в блок и выходом из него. Часто они еще и более удобочитаемы.**
- **Используйте** $foo = $a || $b || $c. **Это значительно быстрее (и короче), чем**

```
if ($a) {
    $foo = $a;
}
elsif ($b) {
    $foo = $b;
}
elsif ($c) {
    $foo = $c;
}
```

Аналогично устанавливайте значения по умолчанию с помощью

```
$pi ||= 3;
```

- **Группируйте все проверки с одинаковой начальной строкой.** Проверяя строки на различные префиксы при помощи какой-либо конструкции, напоминающей структуру оператора switch, группируйте шаблоны /^a/, шаблоны /^b/ и т.д.
- **Не проверяйте строки, про которые заранее известно, что они не содержат интересующих вас соответствий.** Используйте last или elsif, чтобы избежать «проваливания» в следующую ветку оператора switch.
- **Используйте специальные операторы вроде study, логические операции над строками, форматы pack "u" и unpack "%".**
- **Помните о хвосте, который виляет собакой.** Ошибочные команды – что-то вроде (<STDIN>)[0] – могут нагрузить Perl лишней работой. В соответствии с философией UNIX, Perl предоставляет пользователю веревку, достаточно длинную, чтобы на ней можно было повеситься.
- **Выносите операции за циклы.** Оптимизатор не пытается вынести инвариантный код из циклов. Он рассчитывает на вашу сообразительность.
- **Строки могут быть быстрее массивов.**
- **Массивы могут быть быстрее строк.** Все зависит от того, предполагается ли повторно использовать строки или массивы и какие операции предполагается выполнять. Если требуется существенно изменять каждый элемент, лучше использовать массивы, а когда необходимо изменять лишь некоторые элементы, лучше предпочесть строки. Но вы должны попробовать и убедиться в этом сами.
- **Переменные my быстрее, чем переменные local.**
- **Сортировка по созданному вручную массиву ключей может оказаться быстрее, чем использование замысловатой подпрограммы сортировки.** Значение массива обычно сравнивается несколько раз, поэтому если подпрограмма сортировки должна выполнять массивные расчеты, лучше вынести эти вычисления в отдельный проход перед фактической сортировкой.
- **tr/abc//d быстрее, чем s/[abc]//g, если выполняется удаление символов.**
- **Функция print с разделителем-запятой может оказаться быстрее конкатенации строк.** Например, код

```
print $fullname{$name} . " has a new home directory " .
    $home{$name} . "\n";
```

должен склеить вместе два хеша и две фиксированные строки, прежде чем передать их процедурам вывода низкого уровня, тогда как:

```
print $fullname{$name}, " has a new home directory ",
    $home{$name}, "\n";
```

не должен. С другой стороны, в зависимости от значений и архитектуры конкатенация может оказаться быстрее. Проверьте сами.

- **Предпочитайте join("", ...) ряду операций конкатенации строк.** Множественная конкатенация может привести к многократному копированию строк в памяти. Оператор join позволяет избежать этого.

- split с фиксированной строкой обычно быстрее, чем split с шаблоном. Поэтому используйте split(/ /, ...) вместо split(/ +/, ...), если знаете, что пробел будет только один. Однако шаблоны /\s+/, /^/ и / / специально оптимизированы, как и особый split по пробельным символам.
- Упреждающее увеличение массива или строки может сберечь время. По мере роста строк и массивов Perl увеличивает их, размещая в памяти новый экземпляр с некоторым запасом для роста и копируя прежние значения. Упреждающее увеличение строки оператором x или массива установкой значения $#array способны предотвратить такие накладные расходы и уменьшить фрагментацию памяти.
- Не применяйте undef к длинным строкам и массивам, если они будут повторно использоваться с теми же целями. Это помогает предотвратить повторное выделение памяти, когда строку или массив потребуется повторно увеличить.
- Предпочитайте запись "\0" x 8192 вызову unpack("x8192",()).
- system("mkdir ...") может оказаться быстрее для набора каталогов, если системный вызов *mkdir* недоступен.
- Избегайте применения функции eof, если возвращаемые значения уже указывают на конец файла.
- Кэшируйте записи из файлов, которые могут многократно использоваться (например, *passwd* и *group*). Особенно важно кэшировать записи, полученные по сети. Например, для кэширования значений, возвращаемых gethostbyaddr при преобразовании числовых адресов (таких как 204.148.40.9) в имена (такие как «www.oreilly.com»), можно использовать что-то вроде:

```
sub numtoname {
    local ($_) = @_;
    unless (defined $numtoname{$_}) {
        my(@a) = gethostbyaddr(pack("C4", split(/\./)),2);
        $numtoname{$_} = @a > 0 ? $a[0] : $_;
    }
    return $numtoname{$_};
}
```

- Избегайте лишних системных вызовов. Вызовы операционной системы обходятся довольно дорого. Поэтому, например, не вызывайте оператор time, если можно обойтись кэшированным значением $now. Используйте специальный дескриптор файла _, чтобы избежать ненужных вызовов *stat*(2). В некоторых системах даже простейший системный вызов может быть связан с выполнением тысяч инструкций.
- Избегайте ненужных вызовов system. Функция system запускает подпроцесс, чтобы выполнить указанную программу или, что еще хуже, может вызвать интерпретатор команд. Это легко может обернуться выполнением миллионов инструкций.
- Остерегайтесь запуска подпроцессов, если это происходит часто. Запуск одного процесса *pwd*, *hostname* или *find* не сильно вам повредит – в конце концов, интерпретатор занимается запуском подпроцессов постоянно. Можете не верить, но мы иногда поощряем подход «ящика с инструментами».

- Самостоятельно отслеживайте рабочий каталог, избегая многократных вызовов *pwd*. (Для этого имеется стандартный модуль. См. Cwd в главе 28.)
- Избегайте использования в командах метасимволов интерпретатора, если это уместно, передавайте в system и exec готовые списки.
- Устанавливайте «липкий» (sticky) бит на интерпретатор Perl в системах, не поддерживающих подкачку по запросу:

```
chmod +t /usr/bin/perl
```

- Заменяйте вызовы system неблокирующими вызовами open с каналами. Читайте ввод по мере его поступления, не ожидая, пока порожденный процесс завершится.
- Используйте асинхронную обработку событий (AnyEvent, Coro, POE, Gearman и другие), чтобы решать несколько задач одновременно. Современные компьютеры обычно имеют несколько процессоров, по несколько ядер на каждый. Некоторые события могут просто ждать ответа от сети, блокируя программу и не давая ей заниматься другими делами. А некоторые могут быть обусловлены блокирующими вызовами system.

Эффективность по памяти

- Минимизируйте области видимости переменных, чтобы они не занимали память, когда в них нет потребности.
- Для компактного хранения массива целых чисел можно использовать функцию vec, если целые числа имеют фиксированную ширину. (Целые числа переменной ширины можно хранить в строке UTF-8.)
- Числовые значения предпочтительнее эквивалентных строковых значений – они занимают меньше памяти.
- Используйте substr, чтобы хранить строки постоянной длины в более длинной строке.
- Для компактного хранения массива хешей, если длина ключа и значения фиксированы, применяйте модуль Tie::SubstrHash.
- Используйте __END__ и дескриптор файла DATA, чтобы избежать хранения данных программы одновременно в виде строки и массива.
- Применяйте each вместо keys, если порядок не имеет значения.
- Удаляйте или делайте неопределенными неиспользуемые глобальные переменные.
- Организуйте хранение хешей в каком-нибудь файле DBM на диске.
- Сохраняйте массивы во временных файлах.
- Используйте каналы для передачи обработки другим инструментам. Они очищают свою память по завершении работы.
- Избегайте списочных операций и загрузки файлов целиком в память.
- Избегайте tr///. Для каждого выражения tr/// приходится хранить объемистую таблицу трансляции.

- Не разворачивайте циклы и старайтесь обойтись без подстановки (inline) подпрограмм.
- Используйте `File::Map` для чтения файлов, если их содержимое не требуется изменять (и иногда даже если это необходимо).
- Избегайте рекурсии. Perl не выполняет «хвостовую» оптимизацию, поскольку является динамическим языком. Вы сами должны уметь преобразовывать рекурсию в циклический алгоритм, как это делают языки, способные выполнять «хвостовую» оптимизацию.

Для эффективности программирования

Наполовину идеальная программа, работающая уже сегодня, лучше совершенно идеальной программы, которая будет готова только через месяц. Временные проблемы допустимы.[1] Некоторые из следующих советов противоречат тем, что давались выше.

- Загляните в CPAN, прежде чем писать свой код.
- Загляните в CPAN еще раз. Возможно вы что-то пропустили. Поспрашивайте у окружающих.
- Используйте значения по умолчанию.
- Применяйте клёвые сокращенные ключи командной строки, такие как `-a`, `-n`, `-p`, `-s` и `-i`.
- Используйте `for` в смысле `foreach`.
- Выполняйте системные команды посредством обратных апострофов.
- Используйте `<*>` и тому подобное.
- Старайтесь работать с шаблонами, создаваемыми на этапе выполнения.
- Щедро используйте в своих шаблонах `*`, `+` и `{}`.
- Обрабатывайте целые массивы и загружайте файлы целиком в память.
- Используйте `getc`.
- Используйте `` $` ``, `$&` и `$'`.
- Не проверяйте значения ошибок в `open`, так как `<HANDLE>` и `print HANDLE` ведут себя просто как пустые операции, если им передан неверный дескриптор.
- Не вызывайте `close` для своих файлов – они будут закрыты при очередном вызове `open`.
- Не передавайте подпрограммам аргументы. Используйте глобальные переменные.
- Не давайте имен параметрам подпрограмм. К ним можно обращаться непосредственно как к `$_[EXPR]`.
- Используйте первое, что придет в голову.
- Заставьте кого-нибудь сделать половину работы за вас, выложив недоделанный код на Github (*http://www.github.com*).

[1] Так называемые «технические недоработки», которые не всегда являются чем-то ужасным.

Для эффективности сопровождения

Код, который вы (и ваши коллеги) собираетесь использовать и развивать в будущем, заслуживает большего внимания. Замените выгоды в краткосрочной перспективе преимуществами в долгосрочной перспективе.

- Не используйте значения по умолчанию.
- Используйте `foreach` в смысле `foreach`.
- Используйте осмысленные метки циклов в `next` и `last`.
- Давайте осмысленные имена переменным.
- Давайте осмысленные имена подпрограммам.
- Начинайте строку с того, что важно, используя `and`, `or` и модификаторы команд (такие как `exit if $done`).
- Закрывайте файлы, закончив с ними работу.
- Используйте пакеты, модули и классы, чтобы скрыть детали реализации.
- Реализуйте осмысленный API.
- Передавайте аргументы как параметры процедурам.
- Именуйте параметры процедур с помощью `my`.
- Расставляйте круглые скобки для ясности.
- Расставляйте побольше (полезных) комментариев.
- Внедряйте в код pod-документацию.
- `use warnings`.
- `use strict`.
- Пишите тесты, покрывающие как можно больший процент кода (см. главу 19).

Для эффективности переноса на другие платформы

- Помашите приличными чаевыми перед носом носильщика.
- Избегайте функций, которые не везде реализованы. Проверить, какие из них доступны, можно с помощью `eval`.
- Используйте модуль `Config` или переменную `$^O`, чтобы узнать тип машины.
- Добавляйте инструкции `use v5.xx`, чтобы обозначить требуемую версию Perl.
- Не пользуйтесь новыми возможностями языка только затем, чтобы блеснуть эрудицией.
- Не рассчитывайте, что на неизвестной машине будут успешно выполняться `pack` и `unpack` для чисел с плавающей запятой в родном для нее формате.
- Используйте сетевой порядок байтов (форматы `"n"` и `"N"` для `pack`) при отправке двоичных данных по сети.
- Не посылайте двоичные данные по сети. Посылайте ASCII. Лучше посылать UTF-8. Еще лучше посылать деньги.
- Используйте стандартные или распространенные форматы представления данных, такие как JSON или YAML, для обмена информацией.

- Проверяйте $] или $^V, чтобы узнать, поддерживает ли текущая версия все используемые вами возможности.
- Избегайте применения $] и $^V. Используйте require или use с номером версии.
- Применяйте «хак» eval exec, даже если вы не используете его для обеспечения работоспособности программы на тех редких системах, где интерпретаторы команд не распознают обозначение #!.
- Начинайте программу строкой #!/usr/bin/perl, даже если вы не используете ее.
- Проверяйте варианты команд UNIX. Некоторые программы find, например, не умеют обрабатывать ключ *-xdev*.
- Избегайте команд UNIX, если те же операции можно выполнить встроенными средствами. Команды UNIX не слишком хорошо работают в MS-DOS или VMS.
- Помещайте все свои сценарии и страницы руководства в одну сетевую файловую систему, которая смонтирована на всех ваших машинах.
- Опубликуйте свой модуль в CPAN. Вы получите массу откликов, если он окажется не переносимым на другие платформы.
- Дайте другим возможность помочь вам, используйте общедоступные системы управления версиями, такие как Github (*http://www.github.com*).

Для эффективности использования

Сделать проще жизнь других людей намного сложнее, чем упростить задачу себе.

- Не заставляйте набирать строку за строкой, дайте им возможность применить любимый редактор.
- А лучше используйте графический интерфейс, созданный с помощью такого расширения, как Perl/Tk или Wx, где пользователи смогут контролировать порядок событий.
- Дайте пользователям возможность что-нибудь почитать, пока программа работает.
- Используйте автозагрузку, чтобы *казалось*, будто программа работает быстро.
- Предоставьте возможность получать полезные сообщения в любом приглашении командной строки.
- Выводите сообщение о правилах работы, если пользователь ввел некорректные данные.
- Включайте расширенные примеры в документацию и включайте примеры законченных программ в дистрибутив.
- Показывайте действие по умолчанию в каждом приглашении и, возможно, несколько альтернатив.
- Предусмотрите значения по умолчанию для новичков. Разрешите опытным пользователям изменять значения по умолчанию.
- Используйте для ввода один символ, если это имеет смысл.
- Организуйте взаимодействие с пользователем по образцам, с которыми он (пользователь) знаком.

- Сделайте так, чтобы из сообщения об ошибке было понятно, что именно требуется исправить. Включайте всю относящуюся к делу информацию, в том числе имя файла и код ошибки, например:

```
open(FILE, $file) or die "$0: Невозможно открыть $file для чтения: $!\n"
```

- Используйте `fork && exit`, чтобы отключиться (отсоединиться) от терминала, если оставшаяся часть сценария выполняет пакетную обработку.
- Разрешите передавать аргументы из командной строки или через стандартное устройство ввода.
- Используйте конфигурационные файлы в простом текстовом формате. В CPAN имеется масса модулей, поддерживающих такую возможность.
- Не устанавливайте произвольных ограничений в своей программе.
- Предпочитайте поля переменной длины полям фиксированной длины.
- Используйте сетевые протоколы, ориентированные на текст.
- Посоветуйте всем остальным использовать сетевые протоколы, ориентированные на текст!
- Посоветуйте всем остальным советовать всем остальным использовать сетевые протоколы, ориентированные на текст!!!
- Будьте ленивы во благо других.
- Будьте вежливы.

Стиль программирования

Безусловно, у каждого есть свои предпочтения в плане форматирования исходного текста программы, но существуют некоторые общие принципы, которые позволяют облегчить чтение, понимание и сопровождение программ. Ларри озвучил некоторые общие рекомендации в *perlstyle*, но это всего лишь рекомендации. Возможно, вам также понравятся идеи, изложенные в книгах «Perl Best Practices» и «Modern Perl».

Самое важное – запускать программы с директивами `strict` и `warnings`, если нет веских причин не делать этого. Если потребуется отключить их, используйте директиву `no`, распространив ее действие на как можно меньшую область видимости. Полезными могут также оказаться директивы `sigtrap` и даже `diagnostics`.

В отношении эстетики оформления кода едва ли не единственное, о чем сильно беспокоится Ларри, это чтобы закрывающая фигурная скобка многострочного блока имела отступ и была выровнена по ключевому слову, начинающему конструкцию. Помимо этого, у него есть другие предпочтения, которые не столь критичны. Примерам данной книги (должны быть) свойственны следующие соглашения по форматированию:

- Использование отступов в четыре колонки.
- Открывающая фигурная скобка должна по возможности находиться в той же строке, что и предшествующее ключевое слово; в противном случае фигурные скобки следует выравнивать по вертикали:

```
while ($condition) { # для коротких - выравнивать по ключевым словам
    # какие-то действия
```

```
}

# если условие переносится, выравнивать фигурные скобки по вертикали
while ($this_condition and $that_condition
       and $this_other_long_condition)
{
    # какие-то действия
}
```

- **Ставьте пробел перед открывающей фигурной скобкой многострочного блока.**
- **Короткий блок можно поместить в одной строке вместе с фигурными скобками.**
- **Опустите точку с запятой в коротком однострочном блоке.**
- **Возьмите за правило окружать операторы пробелами.**
- **Окружайте «сложный» индекс (внутри квадратных скобок) пробелами.**
- **Помещайте пустые строки между фрагментами кода, выполняющими разные функции.**
- **Разделяйте закрывающую фигурную скобку и** else **посредством перевода строки.**
- **Не ставьте пробел между именем функции и открывающей круглой скобкой.**
- **Не ставьте пробел перед точкой с запятой.**
- **Ставьте пробел после каждой запятой.**
- **Переносите длинные строки после оператора (но перед** and **или** or**, даже в виде** && **и** ||**).**
- **Выравнивайте соответствующие элементы по вертикали.**
- **Опускайте избыточные символы пунктуации, если при этом не страдает ясность.**

У Ларри есть обоснования для каждого из этих пристрастий, но он не утверждает, что у всех остальных голова работает (или не работает) так же, как у него.

Вот несколько еще более существенных соображений относительно стиля:

- **Если нечто *может быть* сделано определенным образом, это еще не значит, что вы *должны* сделать это именно так. Perl может дать несколько способов сделать что-то, поэтому выберите тот, который лучше всего читается. Например:**

  ```
  open(FOO,$foo) || die "Can't open $foo: $!";
  ```

  лучше чем:

  ```
  die "Can't open $foo: $!" unless open(FOO,$foo);
  ```

  потому что второй способ скрывает основной смысл команды в модификаторе. Опять же,

  ```
  print "Starting analysis\n" if $verbose;
  ```

  лучше, чем

  ```
  $verbose and print "Starting analysis\n";
  ```

  поскольку главное не в том, ввел пользователь *-v* **или нет.**

- Аналогично, если оператор допускает аргументы по умолчанию, это не значит, что вы должны использовать значения по умолчанию. Они существуют для ленивых программистов, которые пишут программы «одним махом». Если вы хотите, чтобы вашу программу можно было читать, снабжайте ее аргументами.
- Аналогично, *возможность* опускать круглые скобки во многих местах не означает, что это делать обязательно:

```
return print reverse sort num values %array;
return print(reverse(sort num (values(%array))));
```

  Если есть сомнения, ставьте круглые скобки. Как минимум, они позволят кому-нибудь несчастному понажимать клавишу % в *vi*.

  Даже если *у вас* нет сомнений, подумайте о тех, кому придется сопровождать код после вас и кто может поставить круглые скобки в неверном месте.
- Не следует прибегать к специальным уловкам, чтобы выйти из цикла в начале или конце блока. Perl предоставляет оператор last, обеспечивающий возможность выхода в любой точке. Отступ сделает его более заметным:

```
LINE:
    for (;;) {
            statements;
        last LINE if $foo;
            next LINE if /^#/;
            statements;
    }
```

- Не бойтесь использовать метки циклов – они служат улучшению читаемости, а также позволяют предусмотреть выходы из многоуровневых циклов. См. только что приведенный пример.
- Избегайте применения grep, map и обратных апострофов в пустом контексте, т.е. если возвращаемые ими значения просто отбрасываются. У всех этих функций есть возвращаемые значения, поэтому используйте их. В противном случае следует предпочесть цикл foreach или функцию system.
- Для лучшей переносимости кода, использующего функции, которые могут быть не реализованы в некоторых системах, проверьте конструкцию в eval, чтобы убедиться, что вызов будет успешным. Зная номер версии или уровень исправлений (patch level), обеспечивающие поддержку требуемой возможности языка, можно проверить $] ($PERL_VERSION в модуле English), чтобы узнать, присутствует ли эта возможность. Модуль Config также позволяет запросить значения, которые определила программа *Configure* при установке Perl.
- Давайте переменным и функциям мнемонические имена. Если вы не можете вспомнить, что это за мнемоника такая, у вас проблема.
- Хотя короткие идентификаторы типа $gotit не вызывают возражений, для разделения слов лучше использовать символ подчеркивания. Как правило, значительно легче прочесть $var_names_like_this, чем $VarNamesLikeThis, особенно тем, для кого английский не является родным языком. Кроме того, такое же правило относится к $VAR_NAMES_LIKE_THIS.

  Имена пакетов иногда являются исключениями из этого правила. Perl неформально резервирует имена модулей в нижнем регистре для модулей директив,

таких как integer и strict. Имена других модулей должны начинаться с заглавной буквы и содержать символы разных регистров, но, возможно, не должны включать символы подчеркивания из-за ограничений длины имен в примитивных файловых системах.

- Может оказаться целесообразным использовать регистр букв для указания области видимости или природы переменных. Например:

```
$ALL_CAPS_HERE  # только константы (бойтесь конфликтов с переменными Perl!)
$Some_Caps_Here # глобальные/статические переменные пакета
$no_caps_here   # переменные области видимости функции my() или local()
```

  По различным неочевидным причинам имена функций и методов лучше всего служат, когда полностью записаны в нижнем регистре. Например, $obj->as_string().

  Добавив ведущий символ подчеркивания, можно указать, что переменная или функция не должны использоваться вне пакета, в котором они определены. (Perl не требует соблюдения этого правила, это просто форма документирования.)

- Если встретилось действительно сложное регулярное выражение, используйте модификатор /x и добавьте пробельные символы, чтобы оно меньше походило на белый шум.
- Не используйте косую черту в качестве разделителя, если в регулярном выражении уже слишком много прямых и обратных косых черт.
- Не используйте кавычки в качестве разделителя, если в строке уже есть какие-либо кавычки. Примените псевдофункции q//, qq// или qx//.
- Операторы and и or избавляют от необходимости заключать в круглые скобки списочные операторы и позволяют отказаться от избыточных операторов пунктуации типа && и ||. Вызывайте свои подпрограммы, как если бы они были функциями или списочными операторами, чтобы избежать лишних амперсандов и круглых скобок.
- Используйте внедренные документы вместо многократного повторения команды print.
- Выравнивайте однотипные элементы по вертикали, особенно если они слишком длинны и не умещаются в одной строке:

```
$IDX = $ST_MTIME;
$IDX = $ST_ATIME    if $opt_u;
$IDX = $ST_CTIME    if $opt_c;
$IDX = $ST_SIZE     if $opt_s;
mkdir($tmpdir, 0700)|| die "can't mkdir $tmpdir: $!",
chdir($tmpdir)      || die "can't chdir $tmpdir: $!";
mkdir("tmp", 0777)  || die "can't mkdir $tmpdir/tmp: $!":
```

- Истина, которую мы повторим трижды:

  Обязательно проверяйте значения, возвращаемые системными вызовами.

  Обязательно проверяйте значения, возвращаемые системными вызовами.

  ОБЯЗАТЕЛЬНО ПРОВЕРЯЙТЕ ЗНАЧЕНИЯ, ВОЗВРАЩАЕМЫЕ СИСТЕМНЫМИ ВЫЗОВАМИ!

Сообщения об ошибках должны выводиться в STDERR и указывать, какая программа создала проблему, какой вызов при этом выполнялся и с какими аргументами. Важно, чтобы сообщения об ошибках для системных вызовов содержали стандартные системные сообщения. Вот простой, но показательный пример:

```
opendir(D, $dir) || die "Невозможно выполнить opendir $dir: $!"
```

Обязательно проверяйте возвращаемые значения.

- Выравнивайте транслитерации, когда это имеет смысл:

```
tr [abc]
   [xyz];
```

- Думайте о повторном использовании кода. Зачем тратить умственные усилия на одноразовый сценарий, если что-нибудь подобное может понадобиться потом снова? Попробуйте обобщить свой код. Рассмотрите возможность создания модуля или класса объектов. Заставьте свой код чисто выполняться в области действия `use strict` и `-w`. Рассмотрите возможность бесплатного распространения своего кода. Попробуйте изменить свой взгляд на мир. Попробуйте... а-а, не обращайте внимания.
- Используйте модуль `Perl::Tidy` для форматирования кода и модуль `Perl::Critic` для поиска возможных программных ошибок.
- Будьте последовательны.
- Будьте вежливы.

Беглый разговор на Perl

Мы коснулись нескольких идиом в предшествующих разделах (не говоря уже о предыдущих главах), но есть много других идиом, которые часто встречаются в коде опытных программистов на Perl. Говоря об идиоматическом Perl в данном контексте, мы имеем в виду не набор произвольных выражений с окаменевшими значениями. Скорее мы имеем в виду код, который показывает понимание общего строя языка, что и когда вам может сойти с рук и что вам это даст.

Мы не рассчитываем перечислить все идиомы, которые могут вам встретиться, – для этого потребовалась бы еще одна такая же большая книга. Или две. (См. например «Perl Cookbook»[1]). Но вот несколько важных идиом, где «важных» означает «вызывающих внезапную истерику у человека, считающего, что ему известно, какими должны быть языки программирования».

- Используйте `=>` вместо запятой, если это улучшит читаемость:

```
return bless $mess => $class;
```

Что означает «освятить эту «мешанину» (mess) в указанный класс». Остерегайтесь применять `=>` после слова, которое не должно автоматически заключаться в кавычки:

---

[1] Т. Кристиансен, Н. Торкингтон «Perl: библиотека программиста». – Пер. с англ. – СПб.: Питер, 2001.

```
sub foo () { "FOO" }
sub bar () { 'BAR' }
print foo => bar; # выведет fooBAR, а не FOOBAR;
```

Хорошо также использовать => рядом с литеральной запятой, разглядеть которую может быть нелегко:

```
join(", " => @array);
```

Perl дает возможность решать задачи несколькими способами, чтобы вы могли проявить свои творческие возможности. Проявляйте их!

- Используйте местоимение в единственном числе для улучшения читаемости:

```
for (@lines) {
    $_ .= "\n";
}
```

Переменная $_ является в Perl вариантом личного местоимения. Поэтому приведенный код означает «возьмите каждую строку и прикрепите к *ней* символ перевода строки». В наши дни можно даже записать это так:

```
$_ .= "\n" for @lines;
```

Местоимение $_ настолько важно в Perl, что его употребление обязательно в grep и map. Вот способ кэшировать часто встречающиеся результаты выполнения функции expensive:

```
%cache = map { $_ => expensive($_) } @common_args;
$xval = $cache{$x} || expensive($x);
```

- Для дальнейшего улучшения читаемости опустите и местоимение ($_).[1]
- Используйте операторы управления циклами с модификаторами инструкций.

```
while (<>) {
    next if /^=for\s+(index|later)/;
    $chars += length;
    $words += split;
    $lines += y/\n//;
}
```

Этот фрагмент мы использовали для подсчета страниц в данной книге. Когда приходится выполнять много работы с одной и той же переменной, часто лучше читается код вообще без местоимений.

Этот фрагмент демонстрирует также идиоматическое использование next с модификатором для закорачивания цикла.

Переменная $_ всегда служит управляющей переменной цикла в grep и map, но ссылка программы на нее часто является неявной:

```
@haslen = grep { length } @random;
```

Здесь мы выбираем из списка скаляров только те значения, длина которых больше нуля.

---

1 В данном разделе несколько пунктов списка относятся к одному последующему примеру, поскольку некоторые наши примеры иллюстрируют сразу несколько идиом.

- **Используйте for для установки антецедента местоимения:**

```
for ($episode) {
    s/fred/barney/g;
    s/wilma/betty/g,
    s/pebbles/bambam/g;
}
```

Что из того, что в цикле только один элемент? А то, что это удобный способ «настроить» местоимение, т.е. $_. В лингвистике это называется установкой темы (topicalization) и служит не обману, а коммуникации.

- **Неявно ссылайтесь на местоимение во множественном числе, @_.**
- **Используйте операторы управления для установки значений по умолчанию:**

```
sub bark {
    my Dog $spot = shift;
    my $quality  = shift || "yapping";
    my $quantity = shift || "nonstop";
    ...
}
```

Здесь мы неявно используем другое местоимение Perl, @_, которое означает «они, их» (them). Аргументы всегда передаются функции как «они». Оператор shift знает, что нужно выполнить действие над содержимым @_, если она опущена. Он действует так же, как оператор аттракциона в Диснейленде, который выкрикивает «Следующий!», не указывая конкретно, кто будет этим следующим. (Бессмысленно указывать пальцем, потому что имеется очередь.)

Хотя «по происхождению» оператор || является логическим, он позволяет устанавливать значения по умолчанию, поскольку Perl возвращает первое истинное значение. Программисты часто бесцеремонно обращаются с истиной; приведенная строка не даст желаемого результата, если, например, передать значение для quantity, равное нулю. Но если вы не собираетесь назначать $quality или $quantity булево значение, идиома работает прекрасно. Нет смысла впадать в суеверия и расставлять повсюду defined и exists. Просто нужно понять, что она делает. Пока случайно не будет получено ложное значение, все хорошо.

Если, по вашему мнению, ложное значение все-таки может быть получено случайно, используйте оператор «определено-ИЛИ» //:

```
use v5.10;

sub bark {
    my Dog $spot = shift;
    my $quality  = shift // "yapping";
    my $quantity = shift // "nonstop";

}
```

- **Используйте комбинированные операторы с присваиванием, в том числе в качестве операторов управления:**

```
$xval = $cache{$x} ||= expensive($x);
```

Здесь мы вовсе не инициализируем наш кэш. Мы просто полагаемся на то, что оператор ||= вызовет expensive($x) и присвоит результат $cache{$x}, только если

$cache{$x} имеет ложное значение. Результатом будет новое значение, которое получит $cache{$x}. Снова мы бесцеремонно обращаемся с истиной, поскольку если поместить в кэш ложное значение, функция expensive($x) будет вызвана снова. Возможно, программиста это вполне устраивает, поскольку вызов expensive($x) обходится дешево, когда эта функция возвращает ложное значение. А может быть, программист знает, что expensive($x) вообще не способна вернуть ложное значение. А может быть, программист просто неряшлив. Неряшливость можно считать за форму творчества.

- Используйте операторы управления циклом как операторы, а не как инструкции. И...
- Используйте запятые как «маленькие» точки с запятой в небольшом блоке:

```
while (<>) {
    $comments++, next if /^#/;
    $blank++, next    if /^\s*$/;
    last              if /^__END__/;
    $code++;
}
print "comment = $comments\nblank = $blank\ncode = $code\n";
```

Здесь демонстрируется понимание того факта, что модификаторы инструкций модифицируют инструкции, а next является просто оператором. Здесь показано также идиоматическое использование запятой для разделения выражений, во многом сходное с обычным употреблением точки с запятой. (Различие в том, что запятая удерживает два выражения в составе одной инструкции под контролем одного модификатора команды.)

- Используйте возможность управления потоком выполнения в своих интересах:

```
while (<>) {
    /^#/       and $comments++, next;
    /^\s*$/    and $blank++, next;
    /^__END__/ and last;
    $code++;
}
print "comment = $comments\nblank = $blank\ncode = $code\n";
```

Это тот же самый цикл, на этот раз с шаблонами в начале. Проницательный программист на Perl поймет, что этот цикл компилируется в те же внутренние коды, что и предыдущий пример. Модификатор if является просто обратной конъюнкцией and (или &&), а модификатор unless — обратной дизъюнкцией or (или ||).

- Используйте неявные циклы, создаваемые ключами *-n* и *-p*.
- Не ставьте точку с запятой в конце однострочного блока:

```
#!/usr/bin/perl -n
$comments++, next LINE if /#/;
$blank++, next LINE    if /^\s*$/;
last LINE              if /^__END__/
$code++;

END { print "comment = $comments\nblank = $blank\ncode = $code\n" }
```

Это, в сущности, та же программа, что и выше. Мы поместили метку LINE в операторы управления циклом по собственному желанию, а не из-за реальной необходимости, так как неявный цикл LINE, создаваемый ключом *-n*, является самым глубоким вложенным циклом. Мы использовали END, чтобы вывести завершающую команду за пределы неявного основного цикла, так же как в *awk*.

- Применяйте внедренные документы. когда печать становится устрашающей.
- Используйте осмысленный разделитель для внедренного документа:

```
END { print <<"COUNTS" }
comment = $comments
blank = $blank
code = $code
COUNTS
```

Вместо нескольких инструкций вывода программист, свободно пишущий на Perl, использует строку с интерполяцией, занимающую несколько строк. И несмотря на то что ранее мы назвали это Распространенной Ошибкой, здесь мы без зазрения совести пропустили завершающую точку с запятой, потому что она не обязательна в конце блока END. (Если когда-нибудь мы преобразуем внедренный документ в многострочный блок, то вернем точку с запятой на место.)

- Выполняйте подстановку и трансляцию со скалярами «на проходе»:

```
($new = $old) =~ s/bad/good/g;
```

или используйте модификатор /r, чтобы вернуть результат:

```
$new = $old =~ s/bad/good/gr;
```

Поскольку l-значениям можно присваивать значения, вы часто будете встречать конструкции, меняющие значения «на проходе», т.е. во время присваивания. На практике это может предотвратить внутреннее копирование (если мы когда-нибудь соберемся реализовать соответствующую оптимизацию):

```
chomp($answer = <STDIN>);
```

Любая функция, модифицирующая аргумент на месте, может использоваться в приеме «на проходе». Но погодите, это еще не все!

- Не ограничивайтесь изменением скаляров на проходе:

```
for (@new = @old) { s/bad/good/g }
```

Здесь мы копируем @old в @new, изменяя все на ходу (не все сразу, конечно, – блок выполняется многократно, каждый раз с новым значением).

- Передавайте именованные параметры с помощью красивого оператора запятой =>.
- Предоставьте обработку аргументов операции присваивания хешу:

```
sub bark {
    my DOG $spot = shift,
    my %parm = @_,
    my $quality  = $parm{QUALITY}  || "yapping";
    my $quantity = $parm{QUANTITY} || "nonstop";
    ...
}
```

```
$fido->bark( QUANTITY => "once",
             QUALITY  => "woof" );
```

Именованные параметры часто являются роскошью (которую можно себе позволить). А в Perl вы получаете их даром, если не считать стоимость присваивания хешу.

- Повторяйте логические выражения до получения ложного значения.
- Используйте минимальные квантификаторы в шаблонах, если это возможно.
- Используйте модификатор /e для вычисления выражения замены:

```
#!/usr/bin/perl -p
1 while s/^(.*?)(\t+)/$1 . " " x (length($2) * 4 - length($1) % 4)/e;
```

Эта программа исправляет файлы, получаемые от того, кто ошибочно полагает, что может переопределить аппаратную табуляцию так, чтобы она занимала 4 пробела, а не 8. В ней используется несколько важных идиом. Во-первых, идиома 1 while удобна, когда все, что вы хотите делать в цикле, фактически осуществляется в условии. (Perl достаточно сообразителен, чтобы не предупреждать об использовании 1 в пустом контексте.) Необходимо повторять эту подстановку, поскольку каждый раз, заменяя несколько символов табуляции пробелами, приходится пересчитывать позицию колонки следующей табуляции от начала.

Выражение (.*?) ищет кратчайшую строку до первой табуляции, используя минимальный квантификатор (знак вопроса). В данном случае мы могли прибегнуть к обычному жадному поиску * следующим образом: ([^\t]*). Но это срабатывает только потому, что табуляция является одиночным символом, а значит, можно использовать отрицание класса символов, чтобы не «проскочить» мимо первой табуляции. В целом, минимальный поиск значительно элегантнее и сохраняет работоспособность, даже если следующее найденное соответствие окажется длиннее одного символа.

Модификатор /e осуществляет подстановку, используя выражение, а не просто строку. Это позволяет выполнять необходимые вычисления, когда это требуется.

- Сложные подстановки требуют творческого подхода к форматированию и комментариев:

```
#!/usr/bin/perl -p
1 while s{
        ^                   # привязка к началу
        (                   # начало первой подгруппы
            .*?             # поиск минимального числа символов
        )                   # конец первой подгруппы
        (                   # начало второй подгруппы
            \t+             # поиск одной или более табуляций
        )                   # конец второй подгруппы
}
{
    my $spacelen = length($2) * 4;    # учесть полные табуляции
    $spacelen -= length($1) % 4;      # учесть неполные табуляции
    $1 . " " x $spacelen;             # задать правильное число пробелов
}ex;
```

Это, возможно, излишне, но некоторых впечатляет больше, чем предыдущий однострочник. Решайте сами.

- Не бойтесь применять $`, если вам того хочется:

```
1 while s/(\t+)/" " x (length($1) * 4 - length($`) % 4)/e;
```

Это более короткая версия, включающая $`, о которой известно, что она снижает производительность. Но мы используем только ее длину, поэтому не считается, что это плохо.

- Используйте смещения непосредственно из массивов @- (@LAST_MATCH_START) и @+ (@LAST_MATCH_END):

```
1 while s/\t+/" " x (($+[0] - $-[0]) * 4 - $-[0] % 4)/e;
```

Так еще короче. (Если вы не видите здесь массивов, попробуйте найти элементы массивов.) См. @- и @+ в главе 25.

- Используйте eval с возвратом константы:

```
sub is_valid_pattern {
    my $pat = shift;
    return eval { "" =~ /$pat/; 1 } || 0;
}
```

Совершенно необязательно, чтобы оператор eval {} возвращал реальное значение. Здесь всегда возвращается 1, если оператор выполнился до конца. Однако, если шаблон в $pat окажется некорректным, eval перехватит ошибку и вернет undef в логическое условие оператора ||, который превратит его в определенный 0 (из вежливости, поскольку undef – тоже ложное значение, но из-за него кто-то может подумать, что глючит подпрограмма is_valid_pattern, а нам бы этого не хотелось).

- Выполняйте всю грязную работу при помощи модулей.
- Используйте фабрики объектов.
- Применяйте обратные вызовы (callback).
- Используйте стеки для слежения за контекстом.
- Используйте отрицательные индексы для доступа к концу массива или строки:

```
use XML::Parser;

$p = XML::Parser->new(Style => "subs")
setHandlers $p Char => sub { $out[-1] .= $_[1] };

push @out, ""

sub literal {
    $out[-1] .= "C<";
    push @out, ""
}

sub literal_ {
    my $text = pop @out;
    $out[-1] .= $text . ">";
}
...
```

Это отрывок из программы в 250 строк, с помощью которой мы преобразовали XML-версию старой книги Camel обратно в формат pod, чтобы редактировать ее для этого издания в Real Text Editor, перед тем как перевести в формат DocBook.

Во-первых, можно заметить, что мы использовали модуль XML::Parser (из CPAN) для разбора разметки XML, поэтому нам не пришлось думать, как это сделать. Это сразу сократило нашу программу на несколько тысяч строк (в предположении, что мы заново реализовали бы на Perl все, что модуль XML::Parser делает за нас,[1] в том числе трансляцию символов почти любого набора в кодировку UTF-8).

XML::Parser использует идиому высокого уровня, называемую ***фабрикой объектов*** (***object factory***). В данном случае это фабрика парсеров (синтаксических анализаторов). Создавая объект XML::Parser, мы сообщаем, какой стиль интерфейса анализатора нам требуется, и он создается для нас. Это отличный способ создать приложение для тестирования, когда не ясно, какой тип интерфейса окажется лучшим в конечном итоге. Стиль subs является лишь одним из интерфейсов XML::Parser. На самом деле, это один из самых старых интерфейсов и, возможно, даже не самый популярный в настоящее время.

В строке setHandlers показан вызов метода анализатора не в указательной нотации, а в нотации «косвенного объекта», которая, помимо прочего, позволяет опустить круглые скобки вокруг аргументов. В строке также использована идиома именованного параметра, которую мы видели выше.

В этой строке показана еще одна мощная концепция – обратный вызов (callback). Мы не вызываем анализатор для получения следующего элемента, а предлагаем ему вызвать нас. Для именованных тегов XML, таких как <literal>, этот стиль интерфейса автоматически вызывает подпрограмму с таким же именем (или именем с символом подчеркивания на конце для соответствующих закрывающих тегов). Но данные между тегами не имеют имен, поэтому мы устанавливаем обратный вызов Char с помощью метода setHandlers.

Затем мы инициализируем массив @out, играющий роль стека для вывода. Мы помещаем в него нулевую строку, которая означает, что мы не получили никакого текста на данном уровне вложенности тегов (изначально 0).

Именно теперь снова вступает в дело наш обратный вызов. Как только мы встретили текст, он автоматически дописывается в последний элемент массива через идиому $out[-1] в обратном вызове. На уровне внешнего тега $out[-1] является тем же, что и $out[0], поэтому $out[0] содержит всю выводимую информацию. (В конечном итоге. Но сначала надо разобраться с тегами.)

Допустим, в тексте встретился тег <literal>. Тогда вызывается подпрограмма literal, которая дописывает некоторый текст в текущий элемент @out и проталкивает новый контекст в стек @out. Теперь любой текст до закрывающего тега будет добавлен в этот новый конец стека. Наткнувшись на закрывающий тег, мы выталкиваем собранный нами текст из стека @out в переменную $text и дописываем оставшуюся часть преобразованных данных в новый

---

[1] На самом деле XML::Parser служит просто красивой оболочкой вокруг XML-анализатора ***expat*** Джеймса Кларка (***http://expat.sourceforge.net/***).

(то есть старый) конец стека, результатом чего будет трансляция строки XML <literal>*text*</literal> в соответствующую строку pod, C<*text*>.

Подпрограммы для других тегов точно такие же, только другие.

- Чтобы создать пустой массив или хеш используйте my без присваивания.
- Расщепляйте строку по умолчанию по пробельным символам.
- Выполняйте присваивание спискам переменных, чтобы выбрать любое нужное количество элементов.
- Используйте самооживление (автовивификацию) неопределенных ссылок для их создания.
- Автоинкрементируйте неопределенные элементы массивов и хешей, чтобы создать их.
- Используйте автоинкрементирование хеша %seen для определения уникальности.
- Используйте присваивание удобной временной my -переменной в условном операторе.
- Используйте режим автоматической расстановки кавычек в скобках.
- Применяйте альтернативный механизм расстановки кавычек для вставки двойных кавычек.
- Используйте оператор ?: для переключения между двумя аргументами в printf.
- Выравнивайте аргументы printf по их полям %:

```
my %seen;
while (<>) {
    my ($a, $b, $c, $d) = split;
    print unless $seen{$a}{$b}{$c}{$d}++;
}
if (my $tmp = $seen{fee}{fie}{foe}{foo}) {
    printf qq(Saw "fee fie foe foo" [sic] %d time%s.\n"),
                                        $tmp,  $tmp == 1 ? "" : "s";
}
```

Эти девять строк битком набиты идиомами. Первая строка создает пустой хеш, потому что мы ничего ему не присваиваем. Мы производим итерацию по входным строкам, неявно устанавливая «это», т.е. $_, затем используем split без аргументов, которая расщепляет «это» по пробельным символам. Затем выбираем первые четыре слова, выполняя присваивание списку, и отбрасываем все остальные. Далее мы запоминаем первые четыре слова в четырехмерном хеше, что автоматически создает (при необходимости) первые три элемента ссылок и последний элемент счетчика, который инкрементируется оператором автоинкрементирования. (В области действия прагмы warnings автоинкрементирование не выдает предупреждений об использовании неопределенных значений, поскольку автоинкрементирование является допустимым способом определения неопределенных значений.) Затем мы выводим строку, если ранее не встречалась строка, начинающаяся с этих четырех слов, потому что автоинкрементирование является постинкрементированием, которое, помимо увеличения значения хеша, возвращает прежнее значение, если оно имелось.

По окончании цикла мы снова проверяем %seen, чтобы узнать, встретилась ли конкретная комбинация четырех слов. Мы используем то обстоятельство, что литеральный идентификатор можно поместить в фигурные скобки, и он автоматически будет заключен в кавычки. Иначе нам бы пришлось сказать $seen{"fee"}{"fie"}{"foe"}{"foo"}, что жутко долго набирать, даже если за вами не гонится медведь.

Мы присваиваем результат $seen{fee}{fie}{foe}{foo} временной переменной до проверки его в логическом контексте, предоставляемом if. Поскольку присваивание возвращает свое левое значение, мы все же можем проверить его истинность. Увидев my, вы понимаете, что это новая переменная, и мы не проверяем равенство, а выполняем присваивание. Все прекрасно работало бы и без my, а опытный программист на Perl все равно сразу заметил бы, что мы использовали один символ = вместо двух ==. (Малоопытного программиста на Perl это могло бы ввести в заблуждение. Программисты на Pascal любого уровня мастерства будут в ярости.)

Переходя к инструкции printf, можно заметить, что используется форма двойных кавычек qq(). Это дает возможность вставить обычные двойные кавычки и символ перевода строки. Мы могли бы интерполировать переменную $tmp, поскольку это, по существу, строка в двойных кавычках, но мы предпочли дальнейшую интерполяцию через printf. Наша временная переменная $tmp теперь вполне удобна, особенно учитывая, что мы ее не просто интерполируем, но также используем в условии оператора ?:, чтобы выяснить нужно ли использовать слово «time» во множественном числе. Наконец, обратите внимание, что мы выровняли два поля по соответствующим им маркерам % в строке формата printf. Если аргумент слишком длинный, следующий аргумент можно перенести на другую строку, чего нам в этом случае делать не пришлось.

Уф! Достаточно? Есть много других идиом, которые мы могли бы обсудить, но эта книга и так уже достаточно увесиста. Тем не менее мы хотим поговорить еще об одном идиоматическом использовании Perl – создании генераторов программ.

Генераторы программ

Как только люди обнаружили, что могут писать программы, они стали писать программы, которые пишут другие программы. Мы часто называем их *генераторами программ*. (Любители истории, возможно, знают, что аббревиатура RPG означала Report Program Generator (генератор программ-отчетов) задолго до того, как стала означать Role Playing Game – «ролевая игра».) В наше время их, возможно, назвали бы «фабриками программ», но приверженцы генераторов добрались до них первыми и потому дали им название.

Каждый, кому доводилось писать генератор программ, знает, что от него можно получить косоглазие, даже если быть настороже. Проблема проста: значительная часть данных программы выглядит как настоящий код, но им не является (по крайней мере, до поры до времени). Один текстовый файл содержит и фрагменты, которые что-то делают, и очень похожие фрагменты, которые ничего не делают. Perl предлагает разные способы облегчить компоновку исходного кода Perl с исходным кодом на других языках.

(Конечно, эти возможности облегчают написание Perl на Perl, но, нужно думать, сейчас это и без пояснений должно быть ясно.)

Генерирование других языков на Perl

Perl (помимо прочего) – это язык обработки текста, а большинство языков программирования являются текстовыми. Кроме того, отсутствие в Perl произвольных ограничений в совокупности с различными механизмами расстановки кавычек и интерполяции облегчают зрительную изоляцию кода другого языка, который вы формируете. Вот, например, небольшой фрагмент *s2p*, транслятора *sed*-в-*perl*:

```
print &q(<<"EOT");
:       #!$bin/perl
:       eval 'exec $bin/perl -S \$0 \${1+"\$@"}'
:               if \$running_under_some_shell;
:
EOT
```

Здесь вложенный текст оказывается законным в обоих языках, Perl и *sh*. Мы сразу использовали идиому, которая поможет вам не потерять разум при написании генератора программ: прием с помещением символа «шума» и табуляции в начале каждой цитируемой строки, что зрительно отделяет вложенный код, и благодаря чему с первого взгляда можно сказать, что это не тот код, который в действительности выполняется. Одна переменная, `$bin`, интерполируется в многострочную цитату в двух местах, а затем строка передается в функцию, которая отделит двоеточие и табуляцию.

Конечно, не обязательно использовать многострочные цитаты. Часто можно увидеть сценарии CGI, содержащие миллионы команд `print`, по одной в каждой строке. Примерно как приехать в церковь на болиде Формулы 1, но каждому свое... (Мы готовы допустить, что колонка инструкций `print` обладает своей зрительной выразительностью.)

При встраивании большой многострочной цитаты на каком-нибудь другом языке (например, HTML) часто удобно сделать вид, что вы программируете наоборот, включая Perl в другой язык, как это можно сделать в откровенно вывернутых наизнанку языках, таких как PHP:

```
print <<"XML";
    <stuff>
    <nonsense>
    blah blah blah @{[ scalar EXPR ]} blah blah blah
    blah blah blah @{[ LIST ]} blah blah blah
    </nonsense>
    </stuff>
XML
```

Вы можете прибегнуть к любому из этих двух приемов для интерполяции выражений произвольной сложности в длинную строку.

Некоторые генераторы программ не очень похожи на генераторы программ, в зависимости от того, какую часть своей работы они скрывают от вас. В главе 19 «CPAN» мы видели, как можно использовать маленькую программу *Makefile.PL*

для создания *Makefile*. Размер *Makefile* легко может оказаться в 100 раз больше, чем породившая его *Makefile.PL*. Подумайте, от какой муки это избавило ваши пальцы. Или не думайте об этом – в этом, в конце концов, и заключается смысл.

Генерирование Perl в других языках

Perl позволяет легко генерировать программы на других языках, но верно и обратное. Код на Perl можно легко генерировать из других языков, потому что он лаконичный и гибкий. Можно выбрать свои кавычки, чтобы они не мешали механизму цитирования другого языка. Не надо беспокоиться об отступах или о том, где поместить переносы строки, или об обратной косой черте перед обратной косой чертой. Не обязательно заранее определять пакет как одну строку, потому что разрешается многократно входить в пространство имен своего пакета, как только потребуется выполнить еще код из этого пакета.

Еще одна вещь, которая облегчает написание кода Perl в других языках (включая Perl), – это директива `#line`. Perl умеет обрабатывать ее как специальную директиву, изменяющую его представление об имени текущего файла и номере строки. Это полезно в сообщениях об ошибках и предупреждениях, особенно для строк, обрабатываемых `eval` (которая, если вдуматься, и есть код Perl, который пишет код Perl). Синтаксис этого механизма тот же, что используется препроцессором C: когда Perl встречает символ `#` и слово `line`, за которыми следуют число и имя файла, он устанавливает значение `__LINE__` равным числу, а значение `__FILE__` равным имени файла.[1]

Вот несколько примеров, которые можно проверить, непосредственно введя в *perl*. Для обозначения конца файла используется Control-D, что типично для UNIX. Пользователи DOS/Windows и VMS могут ввести Control-Z. Если ваш интерпретатор команд обозначает конец файла другой комбинацией клавиш, используйте ее, чтобы сообщить *perl* об окончании ввода. С другой стороны, всегда можно ввести `__END__`, чтобы сообщить компилятору, что анализировать больше нечего.

Здесь встроенная функция Perl `warn` выведет новые имя файла и номер строки:

```
% perl
# line 2000 "Odyssey"
# the "#" on the previous line must be the first char on line
warn "pod bay doors"; # or die
^D
pod bay doors at Odyssey line 2001
```

А здесь исключительная ситуация, вызванная `die` в `eval`, попадет в переменную `$@` (`$EVAL_ERROR`) вместе с новыми временными значениями имени файла и номера строки:

```
# line 1996 "Odyssey"
eval qq{
#line 2025 "Hal"
    die "pod bay doors";
};
```

---

[1] Точнее говоря, он ищет шаблон `/^#\s*line\s+(\d+)\s*(?:\s"([^"]+)")?\s*$/`, при этом в `$1` сохраняется номер следующей строки, а в `$2` – необязательное имя файла в кавычках. (Пустое имя файла оставит значение `__FILE__` неизменным.)

```
print "Problem with $@";
warn "I'm afraid I can't do that";
^D
Problem with pod bay doors at Hal line 2025.
I'm afraid I can't do that at Odyssey line 2001
```

Здесь видно, что директива `#line` действует только на текущую единицу компиляции (файл или `eval` *`STRING`*) и после окончания компиляции этой единицы автоматически восстанавливаются прежние установки. Благодаря этому можно организовать собственные сообщения внутри `eval` *`STRING`* или `do` *`FILE`*, не затрагивая оставшуюся часть программы.

Одним из первых препроцессоров Perl был транслятор *sed*-в-*perl*, *s2p*. Более того, Ларри отложил выпуск первой версии Perl, чтобы завершить работу над *s2p* и *awk*-в-*perl* (*a2p*), поскольку полагал, что с ними Perl будет лучше принят. Ну, может быть, они и сыграли свою роль.

Подробнее об этом сказано в электронной документации, а также в трансляторе *find2perl*.

Фильтры исходного кода

Если можно написать программу для трансляции чего угодно в Perl, почему бы не дать возможность вызова этого транслятора из Perl?

Понятие фильтра исходного кода возникло в связи с мыслью, что сценарий или модуль должны уметь расшифровывать себя на лету, например:

```
#!/usr/bin/perl
use MyDecryptFilter;
@*x$]`0uN&k^Zx02jZ^X{.?s!(f;9Q/^A^@~~8H]|,%@^P:q-=
...
```

Но эта идея получила дальнейшее развитие, и теперь можно определить фильтр для любого преобразования исходного кода. Соединив это с действием ключа *-x*, о котором говорилось в главе 17, можно получить общий механизм, позволяющий загрузить любой программный фрагмент из сообщения и выполнить его, независимо от того, написан он на Perl или нет.

С помощью модуля `Filter` из CPAN сейчас можно делать даже такие вещи, как программировать на Perl в *awk*:

```
#!/usr/bin/perl
use Filter::exec "a2p";        # транслятор awk-в-perl
1.30 { print $1 }
```

Это определенно можно отнести к разряду идиом. Но мы ни в коей мере не будем утверждать, что это распространенный прием программирования.

22

Переносимость программ Perl

Мир, в котором существует только одна операционная система, делает переносимость тривиальной, а жизнь скучной. Нам больше по душе обширный генофонд операционных систем, если только экосистема не разделяется однозначно на хищников и их добычу. Perl работает на десятках операционных систем, и, поскольку программы Perl не зависят от платформы, одна и та же программа без модификаций может выполняться на всех этих системах.

Ну, это почти так. Perl старается предоставить программисту как можно больше возможностей, но, если использовать особые функции конкретной операционной системы, это неизбежно ограничит переносимость программы на другие системы. В этом разделе мы дадим некоторые указания относительно написания переносимого кода на Perl. Решив, до какой степени вам необходима переносимость, вы будете знать, где проходят границы, и сможете остаться в их пределах.

Если взглянуть на это с другой стороны, написание переносимого кода обычно означает умышленное ограничение выбора. Естественно, это требует дисциплины и готовности к жертвам, двух качеств, обычно несвойственных программистам на Perl.

На странице *perlport* справочного руководства перечислены платформы, более не поддерживаемые Perl. Среди них, например, Mac OS 9 (Classic) и Windows 95, 98, ME, NT4. Они не просто не поддерживаются – из исходного кода языка Perl был удален весь программный код, обеспечивавший их поддержку. Поэтому, в зависимости от используемой версии Perl, может статься, что вы не сможете поддерживать эти системы. Для поддерживаемых систем, обладающих характерными особенностями и отклонениями, созданы отдельные страницы в справочном руководстве, перечисленные в табл. 22.1.

Таблица 22.1. Страницы справочного руководства для отдельных систем

Страница справочного руководства			
perlaix	*perlfreebsd*	*perlnetware*	*perlsymbian*
perlamiga	*perlhaiku*	*perlopenbsd*	*perltru64*

Страница справочного руководства

perlbeos	*perlhpux*	*perlos2*	*perluts*
perlbs2000	*perlhurd*	*perlos390*	*perlvmesa*
perlce	*perlirix*	*perlos400*	*perlvms*
perlcygwin	*perllinux*	*perlplan9*	*perlvos*
perldgux	*perlmacos*	*perlqnx*	*perlwin32*
perldos	*perlmacosx*	*perlriscos*	
perlepoc	*perlmpeix*	*perlsolaris*	

Имейте в виду, что не все программы на Perl должны быть переносимы. Нет оснований не использовать Perl в качестве связующего кода для инструментов UNIX или для создания прототипов приложений Macintosh, или для управления реестром Windows. Если есть смысл пожертвовать переносимостью, сделайте это.[1] В целом заметьте, что понятия ID пользователя, «домашнего» (home) каталога и даже состояния регистрации существуют только на многопользовательских платформах.[2]

Специальная переменная `$^O` сообщает, в какой операционной системе был скомпилирован Perl. Она служит для ускорения кода, которому в противном случае пришлось бы использовать `Config` для получения тех же данных через `$Config{osname}`. Даже если `Config` был загружен с другими целями, вы все же сэкономите на издержках поиска в связанном хеше. Можно также использовать модули `Devel::AssertOS` и `Devel::CheckOS` из CPAN, предоставляющие дополнительные возможности.

Более подробные сведения о платформе можно получить, просмотрев остальную часть хеша `%Config`, который предоставляется стандартным модулем `Config`. Например, чтобы проверить, поддерживает ли платформа вызов `lstat`, можно выяснить значение `$Config{d_lstat}`. Полное описание имеющихся переменных можно найти в электронной документации по `Config`, а описание поведения встроенных функций Perl на различных платформах можно найти на странице руководства *perlport*. Вот функции Perl, действие которых больше всего зависит от платформы (подробности смотрите в странице *perlport*):

-X (проверка файлов), `accept`, `alarm`, `bind`, `binmode`, `chmod`, `chown`, `chroot`, `connect`, `crypt`, `dbmclose`, `dbmopen`, `dump`, `endgrent`, `endhostent`, `endnetent`, `endprotoent`, `endpwent`, `endservent`, `exec`, `fcntl`, `fileno`, `flock`, `fork`, `getgrent`, `getgrgid`, `getgrnam`, `gethostbyaddr`, `gethostbyname`, `gethostent`, `getlogin`, `getnetbyaddr`, `getnetbyname`, `getnetent`, `getpeername`, `getpgrp`, `getppid`, `getpriority`, `getprotobyname`, `getprotobynumber`, `getprotoent`, `getpwent`, `getpwnam`, `getpwuid`, `getservbyport`, `getservent`, `getservbyname`, `getsockname`, `getsockopt`, `glob`, `ioctl`, `kill`, `link`, `listen`, `lstat`, `msgctl`, `msgget`, `msgrcv`, `msgsnd`, `open`, `pipe`, `qx`, `readlink`, `readpipe`, `recv`, `select`,

---

[1] Не всякий разговор обязательно корректен в любой культуре. Perl старается дать хотя бы один способ сделать Все Правильно, но не пытается навязать вам это. В этом отношении Perl более напоминает ваш родной язык (язык вашей матери), а не язык вашей няни.

[2] Впрочем, понятие «пользователь» в последнее время стало более расплывчатым, потому что даже системы, предназначенные для работы с одним человеком, могут иметь несколько «пользователей».

semctl, semget, semop, send, sethostent, setgrent, setnetent, setpgrp, setpriority, setprotoent, setpwent, setservent, setsockopt, shmctl, shmget, shmread, shmwrite, shutdown, socket, socketpair, stat, symlink, syscall, sysopen, system, times, truncate, umask, utime, wait, waitpid

Перевод строки

В большинстве операционных систем строки файлов завершаются одним или двумя символами, уведомляющими о конце строки. Символы эти различны в разных системах. UNIX традиционно использует \012 (т. е. восьмеричный ASCII-код этого символа равен 12), в системах, ведущих родословную от DOS, используется последовательность \015\012, а старые Маки используют \015. В Perl для представления «логического» перевода строки, независимо от платформы, служит \n. В Perl для DOS \n обычно означает \012, но при обращении к файлу в «текстовом режиме» он транслируется в значение \015\012 или из него, в зависимости от того, читается файл или записывается. UNIX делает то же самое на терминалах в каноническом режиме. Обычно \015\012 обозначается как CRLF.

Поскольку DOS различает текстовые и двоичные файлы, реализации Perl для DOS имеют ограничения при использовании seek и tell с файлами в «текстовом режиме». Для получения оптимальных результатов выполняйте seek только по позициям, получаемым от tell. Однако в случае применения встроенной функции Perl binmode с дескрипторами файлов, как правило, от вызовов seek и tell никакого вреда не случается.

Распространенным заблуждением при программировании сокетов является мнение, что \n всюду означает \012. Во многих протоколах Интернета определены значения \012 и \015, а значения Perl \n и \r ненадежны, так как зависят от операционной системы, где выполняется код:

```
print SOCKET "Hi there, client!\015\012"; # правильно
print SOCKET "Hi there, client!\r\n";     # неправильно
```

Однако использование \015\012 (или \cM\cJ, или \x0D\x0A, или даже v13.10) может быть утомительным и портить внешний вид кода, а также смущать тех, кто занимается его сопровождением. Модуль Socket предоставляет кое-какие Правильные Вещи (для тех, кому они нужны):

```
use Socket qw(:DEFAULT :crlf);
print SOCKET "Hi there, client!$CRLF"      # правильно
```

При чтении из сокета помните, что по умолчанию значением разделителя записей $/ является \n, а это значит, что придется проделать дополнительную работу, если вы не уверены в том, что будете наблюдать в сокете. Надежный код, выполняющий операции над сокетом, должен распознавать в качестве конца строки как \012, так и \015\012:

```
use Socket qw(:DEFAULT :crlf);
local ($/) = LF;        # не требуется, если $/ уже \012

while (<SOCKET>) {
    s/$CR?$LF/\n/;      # заменить LF или CRLF логическим переводом строки
}
```

Аналогично, код, возвращающий текстовые данные, – например, подпрограмма загрузки веб-страницы – часто должен производить трансляцию символов перевода строки. Обычно достаточно одной строки кода:

```
$data =~ s/\015?\012/\n/g;
return $data;
```

Старшинство байтов и ширина чисел

Для хранения целых и вещественных чисел компьютеры используют различные порядки следования байтов: *прямой* (*big-endians*) и *обратный* (*little-endians*).[1] Кроме того, различается и разрядность чисел (32-разрядные и 64-разрядные числа сегодня встречаются чаще всего). Обычно об этом не надо задумываться. Но если программа посылает двоичные данные через сетевое соединение или записывает на диск, который должен читаться другим компьютером, могут потребоваться некоторые меры предосторожности.

Конфликтующие порядки старшинства байтов могут превратить числа в полную мешанину. Если на архитектуре с обратным порядком следования байтов (например, Intel CPU) число хранится в памяти как 0x12345678 (десятичное 305419896), то на архитектуре с прямым порядком следования байтов (например, CPU Motorola) оно будет прочитано как 0x78563412 (десятичное 2018915346). Чтобы избежать этой проблемы в сетевых соединениях (через сокеты), используйте для `pack` и `unpack` форматы `n` и `N`, которые записывают числа `unsigned short` и `long` в порядке big-endian (называемом также «сетевым» порядком) независимо от платформы.

Выяснить порядок следования байтов на своей платформе можно, распаковав структуру данных, упакованную в родном для платформы формате, например:

```
say unpack("h*", pack("s2", 1, 2));
# '10002000' на, скажем, Intel x86 или Alpha 21064 в режиме little-endian
# '00100020' на, скажем, Motorola 68040
```

Определить порядок можно при помощи любой из следующих инструкций:

```
$is_big_endian    = unpack("h*", pack("s", 1)) =~ /01/;
$is_little_endian = unpack("h*", pack("s", 1)) =~ /^1/;
```

Даже если две системы имеют одинаковый порядок следования байтов, проблемы могут все же возникать при передаче данных между 32-разрядными и 64-разрядными платформами. Единственное подходящее решение – избегать передачи и хранения чисел в двоичном виде. Передавайте и храните числа в текстовом виде или применяйте модули, такие как `Data::Dumper` и `Storable`, которые сделают это за вас. В любом случае лучше использовать текстовые протоколы: они более надежны, проще в сопровождении и расширении, чем двоичные.

Конечно, с появлением XML и Юникода определение текста становится более гибким. Например, между двумя системами, на которых выполняется Perl 5.6.0 (или более новая версия), можно передавать последовательность целых чисел, закодированных как символы в `utf8` (Perl-разновидность UTF-8). Если по обе стороны

---

[1] Порядок следования байтов в машинном слове зависит от архитектуры, часто определен в системе через псевдопеременную `BYTE_ORDER`. – *Прим. науч. ред.*

соединения находятся 64-разрядные архитектуры, можно обмениваться 64-разрядными целыми числами. В противном случае вы ограничены 32-разрядными целыми. Используйте для передачи pack с шаблоном U*, а для приема – unpack с шаблоном U* (см. главу 26).

Файлы и файловые системы

Компоненты пути к файлу разделяются в UNIX символом /, в Windows – символом \, а в старых версиях Mac OS – двоеточием (:). Некоторые системы не поддерживают ни жесткие ссылки (link), ни символические (symlink, readlink, lstat). Некоторые системы обращают внимание на регистр символов в именах, некоторые – нет, а какие-то обращают на это внимание при создании файлов, но не при их чтении. Разные системы могут поддерживать разные наборы символов.

Ниже мы дадим несколько советов относительно создания переносимых программ на языке Perl:

- Модуль File::Basename – еще один «терпимый» к платформам модуль, поставляемый в связке с Perl, расщепляет имя файла на компоненты: базовое имя файла, полный путь к каталогу и расширение файла.

```
use File::Basename;

my $name = basename( $ARGV[0] );
my $dir = dirname( $ARGV[0] );

my( $base, $dir, $suffix ) = fileparse( $ARGV[0], qr/\.[^ ]+\z/ )
```

- Стандартные модули File::Spec предоставляют некоторые функции для навигации по файловой системе и объединения компонентов путей к файлам. Не определяйте пути к файлам, но конструируйте их:

```
use File::Spec::Functions;
chdir( updir() );            # переместиться вверх на один каталог
$file = catfile( curdir(), "temp", "file.txt" );
```

  Последняя строка осуществляет чтение из *./temp/file.txt* в UNIX и Windows или из *[.temp]file.txt* в VMS и запоминает содержимое файла в $file.

- Используйте модуль File::HomeDir из CPAN, который находит специальные домашние каталоги пользователей, определяя тип операционной системы и автоматически конструируя правильный путь.
- Используйте модуль Path::Class из CPAN, который предоставляет объектно-ориентированный интерфейс к модулям File::Spec, упрощая чтение путей в формате одной операционной системы и их преобразование в эквивалентные пути в другой системе.
- Используйте модуль File::Temp, входящий в состав Perl, который позволяет создавать временные файлы или файлы с именами, которые до сих пор не использовались.
- Избегайте файлов с именами из одинаковых букв в различных регистрах, например *test.pl* и *Test.pl*, поскольку некоторые платформы игнорируют регистр символов. Некоторые игнорируют, но сохраняют.

- По возможности соблюдайте в именах файлов соглашение 8.3 (восьмибуквенные имена и трехбуквенные расширения). Часто можно справиться с более длинными именами, если обеспечить их уникальность при протаскивании через отверстие размером 8.3. (Это должно быть легче, чем протащить верблюда через игольное ушко.)
- Старайтесь, чтобы в именах файлов было как можно меньше символов, не являющихся буквенно-цифровыми. Символы подчеркивания часто разрешены, но при этом теряется символ, который помог бы обеспечить уникальность в системах 8.3. (Помните, что по этой причине мы обычно не включаем символы подчеркивания в имена модулей.)
- Нормализуйте имена своих файлов или старайтесь не использовать не-ASCII символы. Поддержка Юникода в именах файлов в разных системах находится на разных уровнях, и отсутствует обобщенный API, который одинаково действовал бы во всех системах. Некоторые символы могут быть допустимы в одних системах, но вызывать ошибку в других.
- Аналогично, применяя модуль AutoSplit, постарайтесь ограничить имена подпрограмм восемью или менее символами и не давайте двум подпрограммам имена из одинаковых букв в разных регистрах. Если требуются более длинные имена подпрограмм, сделайте уникальными первые восемь символов каждого имени.
- Всегда явно указывайте символ < при открытии файла для чтения, иначе в системах, допускающих в именах файлов символы пунктуации, открытие файла, имени которого предшествует символ >, может привести к удалению файла, а открытие файла, имени которого предшествует символ |, может привести к открытию канала. Это вызвано тем, что формат open с двумя аргументами является волшебным и интерпретирует такие символы, как >, < и |, что может быть неправильным. (За исключением тех случаев, когда это правильно.)

```
open(FILE,      $existing_file)  || die $!, # неверно
open(FILE,    "<$existing_file") || die $!; # правильнее
open(FILE. "<", $existing_file)  || die $!  # еще правильнее
```

- Выберите кодировку символов для операций ввода/вывода и укажите ее в документации. А лучше дайте пользователям возможность самим выбирать кодировку. Если вы не знаете, чего хотите, используйте UTF-8. Избегайте кодировки UTF-16, которая может создавать проблемы, связанные с порядком следования байтов.
- Не следует предполагать, что текстовые файлы заканчиваются символом перевода строки. Так должно быть, но иногда об этом забывают, особенно если этому помогает текстовый редактор.

Взаимодействие с системой

Платформы, основанные на графическом интерфейсе пользователя, часто не поддерживают командную строку непосредственно, поэтому для обеспечения работоспособности программ, требующих интерфейса командной строки, необходимо предпринять дополнительные шаги.

Некоторые другие советы:

- На некоторых платформах нельзя удалять или переименовывать файлы, находящиеся в работе, поэтому не забывайте закрывать файлы по завершении работы с ними. Не уничтожайте и не переименовывайте открытый файл. Не выполняйте tie или open с уже связанным или открытым файлом; сначала выполните для него untie или close.
- Не открывайте файл на запись более одного раза одновременно, поскольку некоторые операционные системы в обязательном порядке блокируют файлы, открытые для записи.
- Не полагайтесь на существование в %ENV конкретной переменной среды и не считайте, что данные в %ENV чувствительны к регистру или сохраняют регистр. Не рассчитывайте на наследование семантики UNIX для переменных среды; в некоторых системах они могут быть видимы для всех других процессов.
- Не используйте сигналы или %SIG.
- Старайтесь избегать поиска имен файлов по шаблону. Вместо этого применяйте opendir, readdir и closedir. (В Perl версии 5.6 базовый поиск имен файлов по шаблону стал значительно более переносимым, но некоторые системы все равно могут приходить в раздражение от «юниксизмов» интерфейса по умолчанию, если вы попытаетесь соригинальничать.)
- Не полагайтесь на конкретные номера ошибок или строки, хранящиеся в $!.

Межпроцессные взаимодействия (IPC)

Чтобы повысить переносимость, старайтесь не запускать новые процессы. Это означает, что нужно избегать system, exec, fork, pipe, ``, qx// и open с |.

Основную проблему представляют не сами операторы; команды, запускающие внешние процессы, обычно поддерживаются большинством платформ (хотя некоторые платформы не поддерживают ветвление ни в каком виде). Проблемы, скорее могут возникнуть при вызовах внешних программ, для которых семантика имен, адресов, вывода или аргументов может отличаться от платформы к платформе.

Очень популярно использование кода Perl для открытия канала в *sendmail*, чтобы программы могли отправлять электронную почту:

```
open(MAIL, "|/usr/lib/sendmail -t") || die "cannot fork sendmail: $!";
```

Этот код не будет работать на платформах, где нет *sendmail*. Чтобы создать переносимое решение, используйте для отправки почты модули из CPAN, такие как Mail::Mailer и Mail::Send в дистрибутиве MailTools или Mail::Sendmail.

Функции UNIX System V IPC (msg*(), sem*(), shm*()) не всегда доступны даже на платформах UNIX.

В решении некоторых проблем с внешними командами на разных платформах могут помочь модули IPC::Run, IPC::System::Simple и Capture::Tiny из CPAN.

Внешние подпрограммы (XS)

Код XS обычно можно заставить выполняться на любой платформе, но библиотеки и файлы заголовков могут оказаться недоступными либо сам код XS может

оказаться специфическим для платформы. Если библиотеки и заголовки переносимы, разумно предположить, что и код XS тоже можно сделать переносимым.

При написании кода XS возникает проблема переносимости другого рода: наличие компилятора C на платформе конечного пользователя. Язык C привносит свои собственные проблемы переносимости, и написание кода XS знакомит вас с некоторыми из них. Если писать только на Perl, переносимости добиться легче, потому что процесс конфигурирования Perl изо всех сил старается скрыть от пользователя недостатки переносимости C.[1]

Стандартные модули

В целом стандартные модули (модули, увязанные с Perl) работают на всех платформах. Исключение составляют модули `CPAN.pm` (в настоящее время он устанавливает соединения с внешними программами, которых может не быть), специфические для платформ модули (например, `ExtUtils::MM_VMS`) и модули DBM.

Не существует единого модуля DBM для всех платформ. `SDBM_File` и прочие обычно доступны во всех версиях для UNIX и DOS.

Хорошо, что в любом случае хотя бы один модуль DBM будет доступен, и `AnyDBM_File` будет использовать тот модуль, который сможет найти. При такой неопределенности следует применять только те функции, которые есть во всех реализациях DBM. Например, ограничивайте свои записи размером в 1 Кбайт. Подробности можно найти в документации к модулю `AnyDBM_File`.

Чуть лучше дело обстоит с базой данных SQLite, поддержка которой обеспечивает драйвер `DBD::SQLite` для `DBI`. Это минималистичная и встраиваемая реляционная база данных, находящаяся в открытом доступе (благодаря чему ее можно распространять вместе со своим программным кодом). Она может использоваться в большинстве операционных систем.

Дата и время

По возможности используйте для представления дат формат ISO-8601 ("*YYYY-MM-DD*"). Строки вида "1987-12-18" легко преобразуются в специфические для системы значения с помощью модуля типа `Date::Parse`. Список значений, представляющих время и дату (например, возвращаемый встроенной функцией `localtime`), может быть преобразован в специфическое для системы представление с помощью `Time::Local`.

Встроенная функция `time` всегда возвращает число секунд с начала «эпохи», но операционные системы придерживаются различных мнений относительно ее начала. Во многих системах эпоха началась 1 января 1970 года в 00:00:00 UTC, но в VMS она началась 17 ноября 1858 года в 00:00:00. Поэтому для переносимости времени может потребоваться вычислить смещение начала эпохи:

---

[1] Некоторые маргиналы запускают сценарий Perl *Configure* в качестве дешевого развлечения. Известно даже, что устраиваются «Configure races» между конкурирующими системами, на которых разыгрываются крупные суммы. Такая практика в настоящее время признана незаконной в большинстве цивилизованных стран.

```
require Time::Local;
$offset = Time::Local::timegm(0, 0, 0, 1, 0, 70);
```

Значением $offset в UNIX и Windows всегда будет 0, но в других системах это может быть некоторым бо́льшим числом. $offset можно затем складывать со значением времени UNIX и получать величину, которая в любой системе должна быть одинаковой.

Представлением времени суток и календарной даты в системе можно управлять самыми разными способами. Не рассчитывайте, что часовой пояс хранится в $ENV{TZ}. И даже если хранится, не рассчитывайте, что сможете через эту переменную управлять часовым поясом.

Если вам необходима возможность точного управления датой и временем, используйте модуль DateTime из CPAN.

Интернационализация

Не делайте предположений относительно используемой кодировки или окружения. Вы и все ваши знакомые можете использовать одни и те же настройки, но, как только вы займетесь распространением своего продукта, вы наверняка столкнетесь с большим их разнообразием.

Используйте внутри программы Юникод. Включите поддержку преобразования в другие наборы символов и обратно в интерфейсы своего кода. См. главу 6.

За пределами мира Юникода следует делать как можно меньше допущений о наборах символов и воздерживаться от допущений о порядковых значениях символов. Не рассчитывайте, что символы алфавита имеют последовательные порядковые значения. Буквы в нижнем регистре могут идти как до, так и после букв в верхнем регистре; нижний и верхний регистры могут чередоваться так, что оба символа, a и A, будут предварять b; акцентированные и прочие национальные символы могут чередоваться случайным образом, например ä может предварять b.

Даже в мире Юникода следует помнить обо всех перечисленных особенностях. Существует масса алфавитных символов, порядок следования кодов которых никак не связан с их алфавитным порядком.

Если ваша программа должна действовать в системе POSIX (довольно смелое предположение), дополнительную информацию о национальных настройках POSIX вы можете почерпнуть из страницы руководства *perllocale*. Национальные установки определяют, среди прочего, наборы символов и кодировки, а также форматы даты и времени. Правильное применение национальных установок сделает программу более переносимой или хотя бы более удобной и дружественной для пользователей, родным языком которых не является английский. Но помните, что национальные установки и UNIX пока не очень хорошо сочетаются.

Стиль

Если в программе требуется иметь код, специфический для платформы, постарайтесь локализовать этот код, чтобы облегчить перенос на другие платформы. Для определения платформы воспользуйтесь модулем Config и специальной переменной $^O.

Будьте внимательны с тестами, которые предоставляете вместе со своими модулями и программами. Код модуля может быть полностью переносим, а вот тесты – нет. Это часто происходит, если тесты порождают другие процессы или вызывают внешние программы, которые должны содействовать тестированию, и когда (как было отмечено выше) тесты делают некоторые допущения относительно файловой системы и путей в ней. Следите, чтобы не было зависимости от определенного стиля вывода ошибок, даже при проверке в $! «стандартных» ошибок после системного вызова. Лучше использовать модуль Errno.

Помните, что хороший стиль стоит вне времени и вне культуры, поэтому максимальной переносимости можно добиться, лишь отыскав универсальное в запросах, диктуемых суровой действительностью. Самые рассудительные люди не являются пленниками последних причуд моды; им это не нужно, потому что они, уважая собственную культуру (программистскую или любую другую), не стремятся быть в моде. Мода – это переменная, тогда как стиль – константа.

23

Документация в формате POD

Один из принципов, лежащих в основе архитектуры Perl, гласит, что простые вещи должны быть простыми, а сложные вещи должны быть возможными. Документация должна быть простой.

Perl поддерживает простой формат разметки текста, называемый *pod*, который может использоваться самостоятельно или свободно смешиваться с исходным кодом, образуя встроенную документацию. Pod можно преобразовывать во многие другие форматы для печати или просмотра либо читать как есть, потому что это простой текстовый формат.

Язык разметки pod не так выразителен, как XML, LaTeX, *troff*(1) или даже HTML. Это сделано умышленно: мы пожертвовали выразительностью ради простоты и удобства. Некоторые языки разметки вынуждают автора набирать больше разметки, чем самого текста, что излишне осложняет работу и делает почти невозможным чтение. Хороший формат, как хорошая музыка к кинофильму, становится фоном, который не отвлекает внимание.

Заставить программистов писать документацию почти так же сложно, как заставить их носить галстук. Формат pod задумывался таким простым, чтобы даже программист мог (и стал) на нем писать. Мы не утверждаем, что формата pod достаточно, чтобы написать книгу, хотя для написания этой книги его хватило.

Вкратце о pod

Большинство форматов документов требует, чтобы весь документ соответствовал формату. Pod более терпим: можно встраивать pod в файлы любого типа, полагаясь на *трансляторы pod* в извлечении pod. Некоторые файлы целиком набраны в формате pod. Но в других файлах, особенно в программах и модулях Perl, фрагменты pod могут быть разбросаны всюду, где автор счел необходимым. Perl просто пропускает текст pod при синтаксическом анализе файла перед его выполнением.

Лексический анализатор Perl понимает, что нужно начать пропускать текст, когда вместо инструкции обнаруживает строку, начинающуюся знаком равенства в паре с идентификатором, например:

```
=head1 Here There Be Pods!
```

Этот, а также последующий текст, вплоть до строки, начинающейся с =cut, будет игнорироваться. Так мы получаем возможность свободно смешивать исходный текст программы и документацию, например:

```
=item snazzle

The snazzle() function will behave in the most spectacular
form that you can possibly imagine  not even excepting
cybernetic pyrotechnics.

=cut

sub snazzle {
    my $arg = shift;

}

=item razzle

The razzle() function enables autodidactic epistemology generation

=cut

sub razzle {
    print "Epistemology generation unimplemented on this platform.\n";
}
```

Другие примеры можно найти в любом стандартном модуле Perl или в модуле из CPAN – все они должны содержать документацию в формате pod, и почти все действительно ее содержат, кроме тех, где ее нет.

Поскольку разметка pod распознается и отбрасывается лексическим анализатором Perl, можно посредством подходящей директивы pod быстро закомментировать фрагмент кода произвольного размера. Используйте блок pod =for, чтобы закомментировать один абзац, и пару =begin/=end для более крупного участка. О синтаксисе этих директив pod мы расскажем позже. Помните, однако, что в обоих случаях вы остаетесь в режиме pod, пока не вернетесь в компилятор посредством =cut.

```
print "получил 1\n"

=for commentary
Этот абзац игнорируется всеми, кроме некоего
мифического транслятора "commentary"  После его конца вы
все еще в режиме pod, а не в режиме программы.
print "получил 2\n";
=cut

# ok, вот теперь действительно программа
print "получил 3\n";

=begin comment

print  получил 4\n ;
```

```
все, что здесь,
будет игнорироваться
всеми

print "получил 5\n";

=end comment

=cut

print "получил 6\n";
```

При этом выводится, что получены 1, 3 и 6. Помните, что эти директивы pod нельзя поместить в произвольное место. Их можно поместить только там, где анализатор рассчитывает обнаружить новую инструкцию, а не просто в середину выражения или куда-либо еще.

С точки зрения Perl, вся разметка pod отбрасывается, но с точки зрения трансляторов pod отбрасывается как раз код. Трансляторы pod рассматривают оставшийся текст как последовательность абзацев, разделенных пустыми строками.

Абзацы бывают трех типов: буквальные абзацы (verbatim paragraphs), абзацы команд (command paragraphs) и абзацы прозы (prose paragraphs).

Буквальные абзацы

Буквальные абзацы используются для буквального текста, который должен отображаться как есть. Например, когда требуется включить в документацию фрагменты кода. Буквальный абзац должен иметь отступ, т.е. начинаться с пробела или символа табуляции. Транслятор должен точно воспроизвести его, обычно моноширинным шрифтом, а символы табуляции предполагаются на границах каждых восьми колонок. Специальных управляющих символов форматирования нет, поэтому нельзя делать выделение курсивом или полужирным шрифтом. Символ < означает литерал <, и только его.

Абзацы команд

Все директивы pod начинаются символом = и следующим за ним идентификатором. Далее может располагаться произвольный текст любого объема, с которым директива может поступать по своему усмотрению. Единственное требование заключается в том, что весь текст должен быть написан в один абзац. В настоящее время распознаются следующие директивы (иногда называемые *командами pod*):

=encoding

По умолчанию трансляторы pod предполагают, что исходная разметка pod состоит либо из символов ASCII, либо из символов Latin-1. С помощью этой команды можно изменить кодировку, например, на UTF-8:

```
=encoding uft8
```

=head1
=head2

Директивы =head1, =head2, ... создают заголовки заданного уровня. Остальной текст абзаца рассматривается как описание заголовка. Они аналогичны заго-

ловкам разделов и подразделов .SH и .SS в *man*(7) или тегам <H1>...</H1> и <H2>.. </H2> в HTML. На практике именно в такие конструкции трансляторы преобразуют эти директивы.

=cut

Директива =cut указывает на конец фрагмента pod. (Далее в документе может следовать другой фрагмент pod, но он должен начинаться новой директивой pod.)

=pod

Директива =pod лишь указывает компилятору, что анализ кода должен быть отложен до следующей директивы =cut. Полезна для добавления в документ еще одного абзаца, если текст программы и разметка pod чередуются часто.

=over *NUMBER*
=item *SYMBOL*
=back

Директива =over начинает раздел, который должен интерпретироваться как список с элементами, образуемыми директивой =item. Завершается список директивой =back. Число *NUMBER* (если оно указано) указывает программе форматирования, сколько пробелов поместить в отступ. Возможности некоторых программ форматирования недостаточно богаты, чтобы использовать эту подсказку, а других, наоборот, слишком богаты, поскольку при работе с пропорциональными шрифтами трудно выровнять все в одну линию лишь при помощи пробелов. (Обычно считается, что четыре пробела оставляют достаточно места для маркеров или чисел.)

Фактический тип списка определяется параметром *SYMBOL* в каждом элементе. Вот пример маркированного списка:

```
=over 4

=item *

Mithril armor (кольчуга из мифрила)

=item *

Elven cloak (эльфийский плащ)

=back
```

А вот нумерованный список:

```
=over 4

=item 1

First, speak "friend" (скажи "друг").

=item 2.

Second. enter Moria (войди в Морию).

=back
```

А вот именованный список:

```
=over 4

=item armor()

Описание функции armor()

=item chant()

Описание функции chant()

=back
```

Можно создавать вложенные списки одинаковых или разных типов, но при этом нужно придерживаться некоторых основных правил: не используйте =item вне блока =over/=back; укажите хотя бы одну директиву =item в блоке =over/=back; и, вероятно, самое важное, сохраняйте единообразие типов элементов внутри данного списка. Чтобы создать маркированный список, используйте для каждого элемента директиву =item *; чтобы создать нумерованный список – директивы =item 1., =item 2. и т.д.; чтобы создать именованный список – директивы =item foo, =item bar и т.д. Начав с маркеров или чисел, придерживайтесь их, потому что программам форматирования разрешено использовать первый встретившийся тип =item для определения того, как форматировать список.

Как и во всем, что имеет отношение к pod, качество результата зависит от качества транслятора. Одни трансляторы обращают внимание на конкретные числа (или буквы, или римские цифры), следующие за =item, другие – нет. Например, текущая версия транслятора ***pod2html*** весьма бесцеремонна: она вырезает целиком указатели последовательности, даже не пытаясь выяснить тип списка, а потом помещает весь список между тегами <OL> и </OL>, чтобы браузер мог вывести его как упорядоченный список в HTML. Не следует считать такое поведение нормальным; возможно, оно будет со временем исправлено.

=for *TRANSLATOR*
=begin *TRANSLATOR*
=end *TRANSLATOR*

Директивы =for, =begin и =end позволяют включать особые разделы, которые должны передаваться в неизменном виде, но лишь определенным программам форматирования. Программы, которые узнают в *TRANSLATOR* свое имя или псевдоним, обращают внимание на эти директивы; другие полностью их игнорируют. Директива =for указывает, что оставшаяся часть абзаца предназначена конкретному транслятору.

```
=for html
<p> This is a <flash>raw</flash> <small>HTML</small> paragraph </p>
```

Парные директивы =begin и =end действуют аналогично =for, но принимают не один абзац, а весь текст, заключенный между соответствующими =begin и =end, интерпретируя его как предназначенный для конкретного транслятора. Например:

```
=begin html

<br>Figure 1.<IMG SRC="figure1 png"><br>
```

```
=end html

=begin text

---------------
| foo         |
|         bar |
---------------

^^^^ Figure 1 ^^^^

=end text
```

В числе значений *TRANSLATOR*, которые обычно принимаются программами форматирования: `roff`, `man`, `troff`, `nroff`, `tbl`, `eqn`, `latex`, `tex`, `html` и `text`. Некоторые программы форматирования воспринимают их частично как синонимы. Ни один транслятор не воспринимает `comment` – это просто обычное обозначение того, что должно игнорироваться всеми. Таково же предназначение любого нераспознанного слова. Во время написания этой книги авторы часто оставляли для себя пометки с помощью директивы `=for later`.[1]

Обратите внимание, что `=begin` и `=end` могут быть вложенными, но только в том смысле, что самая внешняя пара заставляет рассматривать все вложенные секции как не-pod, даже если там содержатся другие директивы `=`*word*. Это значит, что, когда транслятор встретит `=begin foo`, он будет либо игнорировать, либо обрабатывать *все* до соответствующей директивы `=end foo`.

Поток текста

Абзацы третьего типа – это просто «поток» текста. То есть если абзац начинается не с пробельного символа или знака равенства, он считается простым абзацем – обычным текстом, набранным с минимальным количеством «украшательств». Перевод строки считается эквивалентом пробела. Основная работа по приданию абзацу должного внешнего вида возлагается на транслятор, поскольку у программистов есть более важные дела. Предполагается, что трансляторы используют некоторые распространенные эвристики (см. раздел «Трансляторы и модули pod» далее в этой главе).

Однако кое-что можно делать явным образом. Внутри обычных абзацев или директив заголовков/элементов списка (но не в буквальных абзацах) можно использовать особые последовательности для настройки форматирования. Эти последовательности всегда начинаются с одной заглавной буквы, за которой следует левая угловая скобка, и простираются до парной (не обязательно следующей) правой угловой скобки. В последовательностях могут содержаться другие последовательности.

В pod определены следующие последовательности:

I<*text*>

Текст, выводимый курсивом, предназначен для выделения названий книг, названий кораблей и ссылок на страницы руководства, например «*perlpod*(1)».

---

[1] В действительности мы написали собственный транслятор pod, использующий собственный подкласс класса `Pod::PseudoPod` и преобразующий исходный текст в формате pod в формат DocBook.

B<*text*>

Текст, выводимый полужирным шрифтом, используется преимущественно для ключей командной строки, а иногда для названий программ.

C<*text*>

Буквальный код, возможно, форматированный моноширинным шрифтом, таким как Courier. Не обязателен для простых элементов, которые транслятор должен распознать как код, но использовать его следует в любом случае.

S<*text*>

Текст с неразрывными пробелами. Часто окружает другие последовательности.

L<*name*>

Перекрестная ссылка на имя:

L<*name*>

Страница руководства.

L<*name/ident*>

Элемент страницы руководства.

L<*name/"sec"*>

Раздел в другой странице руководства.

L<*/"sec"*>

То же самое.

Следующие пять последовательностей те же самые, что и выше, но выводиться будет только *text*, а данные о ссылке будут скрыты, как в HTML:

L<*text|name*>

L<*text|name/ident*>

L<*text|name/"sec"*>

L<*text|"sec"*>

L<*text|/"sec"*>

text не может содержать символы / и | и должен содержать < или > только парами.

F<*pathname*>

Применяется для имен файлов. Обычно выводится так же, как I.

X<*entry*>

Элемент указателя. Как всегда, что делать решает транслятор. Спецификация формата pod это не регламентирует.

E<*escape*>

Именованный символ, аналогично escape-последовательностям HTML:

E<lt>

Литерал < (не обязателен, кроме других внутренних последовательностей и после заглавной буквы).

E<gt>

Литерал > (не обязателен, кроме других внутренних последовательностей).

E<sol>

Литерал / (требуется только в L<>).

E<verbar>

Литерал | (требуется только в L<>).

E<*NNN*>
E<0x*XXXXXX*>

Символ с номером (т.е. с кодом) *NNN* или 0x*XXXXXX* в Юникоде.

E<entity>

Некая нечисловая сущность HTML, например E<Agrave>.

Z<>

Символ нулевой ширины. Удобно помещать перед последовательностями, которые могут интерпретироваться неправильно. Например, если в обычной прозе требуется начать строку знаком равенства, можно написать так:

```
Z<>=can you see
```

или в строке с «From», чтобы почтовая программа не поставила впереди >:

```
Z<>From here on out..
```

Чаще всего, чтобы указать границы последовательностей pod, достаточно одной пары угловых скобок. Однако иногда требуется поместить < или > внутрь последовательности. (Особенно часто это происходит в последовательности C<>, применяемой для оформления фрагментов кода моноширинным шрифтом.) Как всегда в Perl, есть несколько способов сделать это. Один из них – представить закрывающую скобку последовательностью E<*ENTITY*>:

```
C<$a E<lt>=E<gt> $b>
```

В результате получится "$a <=> $b".

Более удобоваримый и, возможно, более простой способ – использование альтернативного набора ограничителей, который не требует escape-последовательностей для угловых скобок. Применение двойных угловых скобок (C<< *stuff* >>) допускается при условии, что непосредственно за открывающим ограничителем и непосредственно перед закрывающим ограничителем имеется пробельный символ. Например, можно написать так:

```
C<< $a <=> $b >>
```

Количество повторяющихся угловых скобок может быть любым, лишь бы они были парными, а пробельный символ непосредственно следовал за последней < слева и непосредственно предшествовал первой > справа. Поэтому можно писать и так:

```
C<<< $a <=> $b >>>
C<<<< $a <=> $b >>>>
```

В любом случае, текст $a <=> $b будет форматирован моноширинным шрифтом.

Лишние пробелы внутри с обеих сторон удаляются, поэтому, если вам нужны пробелы, оставьте их снаружи конструкции. Кроме того, два внутренних участка дополнительных пробелов не перекрываются, поэтому, если в начале текста, заключенного в ограничители, стоят двойные угловые скобки >>, они не воспринимаются как закрывающий разделитель:

```
The C<< >> >> right shift operator
```

Эта строка породит текст «The >> right shift operator.».

Обратите внимание, что pod-последовательности могут быть вложенными. То есть, можно написать "The I<Santa MarE<iacute>a> left port already", чтобы получить «The *Santa María* left port already», или "B<touch> S<B<-t> I<time>> I<file>", чтобы получить «**touch** **-t** *time file*», и это будет работать правильно.

Трансляторы и модули pod

Perl поставляется с несколькими трансляторами pod, которые преобразуют документы в формате pod (или содержащие фрагменты в формате pod) в различные другие форматы. Все должны корректно обрабатывать восьмибитные кодировки символов.

pod2text

Преобразует разметку pod в текст. Обычно на выходе получается 7-разрядный текст ASCII; 8-разрядный, если он был 8-разрядным на входе, а если используются последовательности вида LE<uacute>thien (для *Lúthien*) или EE<auml>rendil (для *Eärendil*), текст на выходе будет иметь кодировку ISO-8859-1 (или UTF-8).

Если имеется файл с разметкой pod, вот самый простой (но, возможно, не самый красивый) способ просмотреть только отформатированные фрагменты pod:

```
% pod2text File.pm | more
```

Но напомним, что разметка pod должна быть читаема и без форматирования.

pod2man

Преобразует разметку pod в формат страницы руководства UNIX, пригодный для просмотра с помощью *nroff*(1) или вывода на печать после обработки *troff*(1). Например:

```
% pod2man File.pm | nroff -man | more
```

или

```
% pod2man File.pm | troff -man -Tps -t > tmppage.ps
% ghostview tmppage.ps
```

и вывод на печать:

```
% lpr -Ppostscript tmppage.ps
```

pod2html

Преобразует разметку pod в HTML-документ, который можно просматривать в браузере (даже в *lynx*):

```
% pod2man File.pm | troff -man -Tps -t > tmppage.ps
% lynx tmppage.html
```

pod2latex

Преобразует разметку pod в формат LaTeX, который затем можно сверстать с помощью этого инструмента.

Трансляторы для преобразования в другие форматы можно найти в CPAN.

Составляя документацию в формате pod, следует держаться ближе к простому тексту, не злоупотребляя элементами разметки. Выбирать визуальное представление текста должен уже транслятор. Это означает, что транслятор должен определять, как создавать парные кавычки, как заполнять текстом и выравнивать абзацы, как подобрать более мелкий шрифт для слов, целиком написанных заглавными буквами, и т. д. Поскольку трансляторы предназначаются для обработки документов Perl, в большинстве своем[1] они должны также распознавать следующие элементы и выводить их надлежащим образом:

- FILEHANDLE
- $scalar
- @array
- function()
- manpage(3r)
- somebody@someplace.com
- http://foo.com/

Perl включает также несколько стандартных модулей для анализа и преобразования pod, в том числе Pod::Checker (и соответствующая ему утилита *podchecker*) для проверки синтаксиса документов pod, Pod::Find для поиска документов pod в деревьях каталогов и Pod::Simple для создания собственных утилит обработки pod. Внутри дистрибутивов для CPAN можно использовать модуль Test::Pod для проверки формата документации, а также Test::Pod::Coverage для проверки наличия описаний всех интерфейсов.

Обратите внимание, что трансляторы pod должны рассматривать только абзацы, начинающиеся директивой pod (это облегчает анализ), в то время как компилятор фактически умеет находить escape-последовательности pod даже в середине абзаца. Отсюда следует, что следующий секретный фрагмент secret stuff будет игнорироваться и компилятором, и трансляторами:

```
$a=3;
=secret stuff
warn "Neither POD nor CODE!?
=cut back
print "got $a\n";
```

Вероятно, не следует полагаться на то, что эта директива warn всегда будет исключаться из документации pod. Не все трансляторы pod хорошо ведут себя в этом отношении, да и компилятор в один прекрасный день может стать разборчивее.

Создание собственных инструментов для работы с pod

Формат pod проектировался таким образом, чтобы в нем, прежде всего, было легко писать. Бесплатное приложение к этому: простота pod дает возможность создавать простые средства для обработки этого формата. Чтобы найти директивы

[1] Если вы создаете транслятор pod общего назначения, а не только для кода на Perl, критерии могут быть иными.

pod, установите разделитель записей в режим абзацев (вероятно, с помощью ключа *-00*) и обращайте внимание только на абзацы, похожие на разметку pod.

Вот, например, простая программа *olpod*, извлекающая общую структуру pod-разметки:

```
#!/usr/bin/perl -l00n
# olpod - outline pod
next unless /^=head/;
s/^=head(\d)\s+/ " " x ($1 * 4 - 4)/e;
print $_, "\n";
```

Если применить ее к исходному тексту данной главы, можно получить нечто вроде:

```
Документация в формате pod
  Вкратце о pod
    Буквальные абзацы
    Абзацы команд
    Поток текста
  Трансляторы и модули pod
  Создание собственных инструментов для работы с pod
  Ловушки pod
  Документирование программ на Perl
```

По правде сказать, приведенная программа не обращала внимания на то, были ли блоки pod корректными. Поскольку разметку pod можно смешивать с любым другим текстом, запуск средств общего назначения для поиска или анализа файла целиком не всегда имеет смысл. Но это не проблема, учитывая, как легко создавать средства для работы с pod. Вот инструмент, который *отличает* фрагменты pod и выводит только их:

```
#!/usr/bin/perl -00
# catpod - cat только pod
while (<>) {
    if (! $inpod) { $inpod = /^=/;            }
    if ($inpod)   { $inpod = !/^=cut/; print; }
} continue {
    if (eof)      { close ARGV; $inpod = 0 }
}
```

Можно использовать эту программу с другой программой или модулем Perl, а затем перенаправить вывод через конвейер в другую программу. Например, если имеется программа *wc*(1)[1] для подсчета строк, слов и символов, можно накормить ее выводом *catpod*, чтобы при подсчете учитывались только фрагменты pod:

```
% catpod MyModule.pm | wc
```

Есть масса ситуаций, в которых pod позволяет писать примитивные инструменты, тривиально используя простой бесхитростный Perl. Теперь у нас в качестве компонента есть *catpod* и можно создать еще один инструмент, который выводит только код с отступами:

---

1 **Ее можно найти в Perl Power Tools из каталога CPAN (*https://www.metacpan.org/release/ppt/*).**

```
#!/usr/bin/perl -n00
# podlit - вывести литеральные блоки с отступом из входных данных pod
print if /^\s/;
```

Как это можно использовать? Например, можно реализовать проверки *perl -wc* для кода, находящегося в документе. Или создать разновидность *grep*(1)[1], которая ищет только в примерах кода:

```
% catpod MyModule.pm | podlit | grep funcname
```

Такая философия инструментов и фильтров, состоящих из взаимозаменяемых (и допускающих независимое тестирование) частей, представляет собой чрезвычайно простой и мощный подход к разработке повторно используемых программных компонентов. Это разновидность лени, состоящая в том, чтобы находить минимальные решения, позволяющие решать повседневные задачи, по крайней мере некоторые.

В других случаях, однако, это может оказаться контрпродуктивным. Иногда написание инструмента с нуля требует больше труда, иногда меньше. Для того, что было нами показано выше, собственное мастерство Perl в обработке текста делает целесообразным применение грубой силы. Но не всегда получается именно так. Развлекаясь с форматом pod, можно заметить, что его директивы легко анализировать, но последовательности иногда оказываются несколько неочевидными. Последовательности могут быть вложены в другие последовательности и использовать ограничители переменной длины, хотя некоторые, с позволения сказать, «недокорректные» трансляторы этого не поддерживают.

Помогая нам избежать необходимости писать весь этот код для синтаксического разбора, лень подсказывает другое решение. Можно воспользоваться стандартным модулем `Pod::Simple`. Сей модуль особенно полезен для сложных задач, когда, например, действительно требуется анализировать внутренние участки абзацев, преобразовывать их в альтернативные выходные форматы и т.д. В сложных случаях с этим модулем легче работать, поскольку в итоге приходится писать меньше кода. Удобство еще и в том, что хитрый синтаксический анализ оказывается уже проделанным. Несомненно, это тот же принцип, что и при использовании *catpod* в конвейере.

Для решения своих задач модуль `Pod::Simple` использует довольно интересный подход. Это объектно-ориентированный модуль иного типа, чем большинство тех, что вы видели в этой книге. Его основная задача не в том, чтобы предоставить объекты для непосредственного использования, а в том, чтобы дать базовый класс, на основе которого можно строить другие классы.

Вы создаете собственный класс, наследующий `Pod::Simple` (или один из его интерфейсов) и реализующий все методы, необходимые для синтаксического анализа разметки pod. Подкласс должен переопределять соответствующие методы, чтобы помочь анализатору вывести все, что вам требуется. Чтобы изменить поведение анализатора в желаемую сторону, достаточно переопределить только часть методов. Возможно, для начала лучше написать транслятор, делающий лишь шаг к желаемому. Ниже приводится подкласс `Pod::Simple::Text`, отыскивающий в до-

[1] А если у вас нет *grep*, см. предыдущую сноску.

кументации фрагменты L<> и создающий концевые сноски для каждой ссылки. Вы должны знать внутреннее устройство базового класса, что является нарушением инкапсуляции, которое мы лишь демонстрируем, но не одобряем:

```
use v5.14;

package Local::MyText 0.01 {
    use parent "Pod::Simple::Text"
    use Data::Dumper;
    my @links;

    sub links {
        $_[0]->{"".__PACKAGE__}{links} //= [];
    }

    sub start_L {
        my($self, $link) = @_;
        push $self->links, $link->{to}[2];
    }

    sub end_L {
        my($self) = @_;
        my $count = @{$self->links};
        $self->{Thispara} .= "[" . $count . "]";
    }

    sub end_Document {
        my($self) = shift;
        while (my($index, $text) = each $self->links) {
            $self->{Thispara} .=
                "$index http://perldoc.perl.org/$text.html\n";
        }
        $self->emit_par;
    }
}

1;
```

Можно даже написать собственную версию программы *pod2text*, загружающую файл и вызывающую подкласс транслятора, но это необязательно, потому что *perldoc* позволяет загружать альтернативные классы форматирования с помощью ключа *-M*:

```
% perldoc -MLocal::MyText some_pod.pod
```

Для следующего фрагмента документации pod:

```
=pod

Если у вас появится желание ознакомиться со спецификацией Perl pod
читайте документацию LZ<><perlpod> или LZ<><perlpodspec>.

=cut
```

получится следующее:

```
Если у вас появится желание ознакомиться со спецификацией Perl pod,
читайте документацию perlpod[1] или perlpodspec[2].

0 http://perldoc.perl.org/perlpod.html
1 http://perldoc.perl.org/perlpodspec.html
```

Этот пример просто изменил порядок интерпретации транслятором спецификации pod. Ниже приводится другой пример, реализующий форматирование буквальных абзацев с помощью Perl::Tidy**:**

```
use v5.14

package Local::MyTidy 0.01 {
    use parent "Pod::Simple::Text";
    use Perl::Tidy;

    sub end_Verbatim {
        my($self) = @_;
        Perl::Tidy::perltidy(
            source      => \ $self->{Thispara},
            destination => \ my $out,
            argv        => [qw/-gnu/],
        ),
        $self->{Thispara} = $out =~ s/^/ /gmr;
        say { $self->{output_fh} } "", $self->{Thispara};
        return;
    }
}

1;
```

Этот класс отформатирует такой код:

```
=encoding utf8

=pod

Это обычный абзац.

#!/usr/bin/perl
use v5.14;
for (@ARGV){
    state $count = 0;
    say $count++, " ", $_
}

Еще один обычный абзац

=cut
```

следующим образом:[1]

1 **Модуль** Perl::Tidy **понимает множество различных параметров, позволяя вам организовать форматирование по своему вкусу.**

```
Это обычный абзац

#!/usr/bin/perl
use v5.14;
for (@ARGV) {
    state $count = 0;
    say $count++, " ", $_,
}

Еще один обычный абзац.
```

Используя такой подход, можно даже расширять формат pod. Например, если у вас появится желание добавить новую команду, это легко сделать (хотя и не рекомендуется). Нужно лишь сообщить анализатору, что новая команда является допустимой. В следующем примере новая команда `V<>` **транслирует свой текст в список кодов символов. Вместо é в список попадет значение (U+00E9). Это достигается установкой флага при входе в команду** `V<>`**, чтобы в** `handle_text` **можно было узнать, что требуется сделать:**

```
use v5.14;
package Local::MyCodePoint 0.01 {
    use parent "Pod::Simple::Text";
    use Data::Dumper;

    sub new {
        my $self = shift;
        my $new = $self->SUPER::new;
        $new->accept_codes("V");
        return $new;
    }

    sub handle_text {
        my($self, $text) = @_;
        $self->{Thispara} .=
        $self->{"".__PACKAGE__}{in_V}
            ? $self->make_codepoints($text)
            : $text;
    }

    sub make_codepoints {
        $_[1] =~ s/(.)/ sprintf "(U+%04X)", ord($1) /ger;
    }

    sub start_V {
        my($self, $text) = @_;
        $self->{"".__PACKAGE__}{in_V} = 1;
    }

    sub end_V {
        my($self, $text) = @_;
        $self->{"".__PACKAGE__}{in_V} = 0;
    }
}

1;
```

С поддержкой новой команды следующий фрагмент pod:

```
=encoding utf8

=pod

V<À> la recherche du temps perdu

=cut
```

превратится в:

```
(U+00C0) la recherche du temps perdu
```

Ловушки pod

Формат pod весьма прост, но все же в некоторых местах можно допустить промахи, сбивающие с толку некоторые трансляторы:

- Крайне легко пропустить закрывающую угловую скобку.
- Крайне легко пропустить замыкающую директиву =back.
- Просто легко случайно вставить пустую строку в середине длинной директивы =for *comment*. Попробуйте использовать вместо нее =begin/=end.
- Если допустить опечатку в одном из тегов в паре =begin/=end, будет «съедена» вся оставшаяся часть вашего файла (имеется в виду только разметка pod). Попробуйте вместо этого применить директиву =for.
- Трансляторы pod требуют, чтобы абзацы разделялись абсолютно пустыми строками, т.е. двумя или более последовательными символами перевода строки (\n). Если в строке есть пробелы или символы табуляции, она не считается пустой. В результате два или более абзаца рассматриваются как один.
- Понятие «ссылки» в pod не определено, и каждый транслятор сам решает, что с ней делать. (Тот, кто начинает подозревать, что в большинстве случаев решение остается за трансляторами, прав.) Трансляторы часто добавляют слова вокруг ссылки L<>, поэтому "L<foo(1)>" может, например, превратиться в «the *foo*(1) manpage». Поэтому не следует определять ссылки в виде "the L<foo> manpage", если требуется, чтобы оттранслированный документ осмысленно читался, иначе может получиться «the the *foo*(1) manpage manpage».

  Если необходим полный контроль над текстом ссылки, используйте формат L<show this text|foo>.

Стандартная программа *podchecker* проверяет синтаксис pod и выводит ошибки или предупреждения. Например, она обнаруживает неизвестные последовательности pod и предположительно пустые строки, содержащие пробельные символы. Кроме того, рекомендуется пропустить документ pod через как минимум два различных транслятора pod и проверить результаты. Некоторые возникающие проблемы могут быть особенностями конкретных трансляторов, и вам решать, будете ли вы искать пути обхода этих особенностей. Ниже приводится небольшой фрагмент в формате pod, имеющий некоторые проблемы:

```
=encoding utf8

=pod
```

```
Это - D<para>

=item • Элемент списка

=cut
```

Программа *podchecker* обнаружит здесь две ошибки и выведет предупреждение о пробеле, невидимом в пустой строке:

```
% podchecker broken.pod
*** ERROR: Unknown interior-sequence 'D' at line 5 in file broken.pod
*** ERROR: =item without previous =over at line 7 in file broken.pod
*** WARNING: line containing nothing but whitespace in paragraph at line 8
in file broken.pod
broken.pod has 2 pod syntax errors

[*** ERROR: Неизвестная последовательность 'D' в строке 5 файла broken.pod
 *** ERROR: =item без предшествующей =over в строке 7 файла broken.pod
 *** WARNING: строка абзаца, не содержащая ничего, кроме пробела, в строке 8
файла broken.pod
broken.pod содержит 2 ошибки синтаксиса pod.]
```

И, как всегда, Все Может Измениться по Прихоти Безымянного Кодоборца.

Документирование программ Perl

Мы надеемся, что читатель документирует свой код независимо от того, является ли он Безымянным Кодоборцем. Если это так, то, вероятно, вы захотите включить в документацию следующие разделы:

=head1 NAME

Название программы или модуля.

=head1 SYNOPSIS

Аннотация назначения модуля.

=head1 DESCRIPTION

«Простыня» документации. (В данном контексте «простыня» – хорошая практика.)

=head1 AUTHOR

Кто вы такой. (Или ваш псевдоним, если вам стыдно за свою программу.)

=head1 BUGS

Что вы делали неправильно (и почему на самом деле вы в этом не виноваты).

=head1 SEE ALSO

Где можно найти дополнительную информацию (чтобы справиться с ошибками в программе).

=head1 COPYRIGHT

Уведомление об авторских правах. Если вы хотите явным образом заявить об авторских правах, можно сказать что-то вроде:

```
Copyright 2013, Randy Waterhouse. All Rights Reserved.
```

Во многих модулях также добавляется:

```
This program is free software. You may copy or
redistribute it under the same terms as Perl itself
[Эта программа распространяется свободно. Вы можете копировать
или распространять ее на тех же условиях, что и Perl]
```

Одно предостережение: помещая документацию в конец файла и используя лексемы `__END__` или `__DATA__`, не забудьте вставить пустую строку перед первой директивой pod:

```
__END__

=head1 NAME

Modern - I am the very model of a modern major module
```

Если перед `=head1` не будет пустой строки, трансляторы pod не обнаружат вашу (обширную, точную и грамотную) документацию.

24

Культура Perl

Эта книга является частью культуры Perl, и мы не можем надеяться, что в нее удастся вместить все, что мы знаем о культуре Perl. Немного истории и немного искусства – кто-то сказал бы «очень немного искусства» – и некоторые факты из жизни сообщества позволят нам раздразнить ваш аппетит. Значительно больше о культуре Perl сказано на *http://www.perl.org*. Или просто познакомьтесь с другими программистами на Perl. Мы не можем сказать, какого сорта люди вам попадутся: едва ли не единственная общая черта характера программистов на Perl – это патологическая готовность прийти на помощь.

История практичности

Чтобы понять, почему Perl получился именно таким (а не каким-то другим), придется сначала разобраться, почему Perl вообще появился на свет. Поэтому стряхнем пыль с нашего старого учебника истории...

Давным-давно, аж в 1986 году, Ларри работал системным программистом в проекте по разработке глобальных сетей с многоуровневой защитой. Он отвечал за структуру, состоявшую из трех VAX и трех Sun на западном побережье, соединенных на скорости 1200 бод через зашифрованную последовательную линию с аналогичной конфигурацией на восточном побережье. Поскольку основной задачей Ларри была поддержка (в проекте он участвовал не как программист, а просто как системный гуру), он смог воспользоваться тремя своими добродетелями (*ленью*, *нетерпеливостью* и *высокомерием*), чтобы разработать и усовершенствовать всевозможные полезные инструменты, например *rn*, *patch* и *warp*.[1] Однажды,

---

[1] Примерно в это время Ларри уловил смысл фразы «feeping creaturism» (В «Новом словаре хакера» Эрика С. Реймонда чудесно сказано, что, во-первых, этот термин происходит от «creeping featurism», а во-вторых, «...четкого определения ... нет, но любой хакер поймет, о чем идет речь». – *Прим. перев.*) в отчаянной попытке оправдать, на основании биологической необходимости, свое непреодолимое стремление «добавить еще одну функцию». В конце концов, если Жизнь Попросту Слишком Сложна, почему то же

как раз когда Ларри разорвал *rn* в клочья и разбросал их по своему каталогу, до него снизошел великий Менеджер и произнес: «Ларри, нам нужна система управления конфигурацией и контроля для всех шести VAX и всех шести Sun. Через месяц. Займись-ка этим!»

И Ларри, который никогда не уклонялся от работы, задал себе вопрос: как лучше всего сконструировать систему управления компьютерами на восточном и западном побережьях, позволяющую видеть отчеты о проблемах, возникших в обоих местах, давать разрешения и осуществлять контроль, и при этом не писать все с нуля. Ответ пришел в виде одного слова: B-news.[1] Ларри взялся за дело и установил на этих машинах программы работы с телеконференциями, добавив две команды управления: «append», чтобы выполнять добавление к существующей статье, и «synchronize», чтобы номера статей были одинаковыми на обоих побережьях. Управление компьютерами предполагалось осуществлять посредством системы управления версиями RCS (Revision Control System), а подтверждения и новые запросы выполнять посредством телеконференций и *rn*. Для начала очень хорошо.

Затем великий Менеджер попросил представить ему отчеты. Сообщения хранились в отдельных файлах на главной машине, а файлы были связаны массой перекрестных ссылок. Сначала Ларри подумал: «Используем *awk*». К несчастью, в то время *awk* не справлялась с открытием и закрытием многих файлов на основе информации в этих файлах. Ларри не захотел писать инструментальное средство для решения специальной задачи. В результате появился новый язык.

Этот новый инструмент изначально не назывался Perl. Ларри обсуждал некоторые названия с коллегами и помощниками – Дэном Фэйджином (Dan Faigin), автором этого повествования, и Марком Биггаром (Mark Biggar), своим родственником, который также очень помог ему в начальной разработке. Ларри рассмотрел и отверг все трех- и четырехбуквенные слова из словаря. На заре своего существования Perl носил, в частности, название «Gloria» в честь возлюбленной (и жены) Ларри. Но затем Ларри решил, что это вызовет ненужную путаницу в семье.

Затем возникло название «Pearl», превратившееся в современное «Perl» отчасти потому, что Ларри встретилось упоминание другого языка с названием PEARL, но главным образом потому, что Ларри слишком ленив, чтобы каждый раз набирать пять букв. И конечно, потому, что Perl подходил на роль слова из четырех букв[2]. (Однако следы прежнего написания можно обнаружить в толковании акронима: «Practical Extraction *And* Report Language».[3])

Ранний Perl не обладал многими возможностями современного Perl. В нем уже были поиск по шаблону и дескрипторы файлов, скаляры и форматы, но было очень мало функций, довольно увечная реализация регулярных выражений, за-

---

будет справедливо и для программ? Особенно для таких программ, как *rn*, которые следовало бы рассматривать как передовые проекты искусственного интеллекта, потому что они уже на грани того, чтобы зачитывать вам новости вслух. Правда, кое-кто скажет, что даже программа *patch* уже слишком *заумна*.

1 То есть второй реализации транспортного программного обеспечения Usenet.

2 В английском языке идиома «four-letter word» обозначает ругательства и бранные слова, многие из которых в английском состоят как раз из четырех букв. – *Прим. ред.*

3 Иногда это называют *обратной аббревиатурой* (*backronym*), поскольку сначала появилось название, а потом его расшифровка.

имствованная из *rn*, и отсутствовали ассоциативные массивы. Объем руководства составлял всего 15 страниц. Но Perl был быстрее, чем *sed* и *awk*, и стал использоваться в других приложениях проекта.

Но Ларри был нужен везде. В один прекрасный день пришел другой великий Менеджер, который сказал: «Ларри, займись поддержкой исследовательских работ». Ларри просто согласился. Он прихватил с собой Perl и обнаружил, что язык превращается в хорошее средство системного администрирования. Он позаимствовал великолепный пакет регулярных выражений Генри Спенсера (Henry Spencer) и перекромсал его так, что Генри старался не думать об этом за едой. Затем Ларри добавил кое-что для себя и кое-что для других. Он опубликовал Perl в сети.[1] Остальное, как говорят, уже история.[2] А развивается она примерно так. Perl 1.0 вышел 18 декабря 1987 года; некоторые до сих пор всерьез отмечают День Рождения Perl в этот день. В июне 1988 последовал Perl 2.0, и Рэндал Шварц (Randal Schwartz) создал свою легендарную подпись «Just Another Perl Hacker» (*JAPH*). В 1989 году Том Кристиансен (Tom Christiansen) представил первый общедоступный учебник Perl на USENIX в Балтиморе. Начиная с версии Perl 3.0, вышедшей в октябре 1989, язык впервые стал распространяться на условиях GNU Public License.

В марте 1990 Ларри пишет первое стихотворение – Perl Poem (см. следующий раздел). Затем в соавторстве с Рэндалом первое издание этой книги – «The Pink Camel» (розовый верблюд); она увидела свет в начале 1991 года.[3] Одновременно появился Perl 4.0; распространявшийся уже на основе двух лицензий, GPL и Artistic License. После выхода Perl 4 Ларри задумал создать улучшенную версию Perl; в 1994 году появилась группа Perl 5 Porters, или просто *p5p*, взвалившая на себя бремя переноса *perl* практически на все платформы, до которых смогла дотянуться. Состав этой группы постоянно меняется, но она до сих пор отвечает за развитие Perl и его поддержку.

Презентация весьма ожидаемого Perl 5 прошла в октябре 1994. Perl переписали полностью, в нем появились объекты и модули. Пришествие Perl 5 даже удостоилось освещения в «The Economist».[4] В 1995 сообщество Perl официально познакомили с архивом CPAN. В 1996 Джон Орвант (Jon Orwant) начал издавать «The Perl Journal». Осенью того же года после длительного вынашивания родилось второе издание этой книги, «The Blue Camel» (голубой верблюд). В 1997 группа известных Perl-активистов основала организацию «The Perl Institute», занимающуюся популяризацией и поддержкой Perl.

---

[1] Что еще более удивительно, Ларри продолжал выпускать новые версии, когда начал работать в Jet Propulsion Lab, затем в NetLabs и Seagate, а затем в O'Reilly & Associates (это небольшая компания, издающая памфлеты о компьютерах и подобной дребедени; сегодня она называется O'Reilly Media).

[2] И раз так, даем историческую справку. Когда началась работа над Perl, *rn* как раз была разобрана «по кирпичику» для капитальной реконструкции. Начав работать над Perl, Ларри не прикоснулся к *rn*. Она по-прежнему разобрана на кусочки. Время от времени Ларри грозится переписать *rn* на Perl, но это он не всерьез.

[3] И называлась «Programming perl», где слово «perl» было набрано строчными буквами.

[4] «В отличие от большинства свободно распространяемых программ, Perl не только работает, но и приносит пользу» – «Electric metre», The Economist, 1 июля 1995 года.

Летом 1997 года в Сан-Хосе, штат Калифорния, прошла первая конференция O'Reilly по Perl «The Perl Conference» (TPC). На этой конференции группа жителей Нью-Йорка организовала первую группу пользователей Perl, получившую название /New York Perl M((o|u)ngers|aniacs)*/. По соображениям удобства это название позднее превратилось в NY.pm и послужило образцом для большинства названий групп пользователей Perl, появившихся позднее. В следующем году, когда эта же группа помогла другим в создании собственных групп пользователей, она превратилась во всемирную группу пользователей Perl (*http://www.pm.org*) и приняла на себя обязанности группы «The Perl Institute».

В 1999 году Кевин Ленцо (Kevin Lenzo) организовал конференцию «Yet Another Perl Conference» (YAPC) в университете Карнеги-Меллона (Carnegie Mellon) в Питсбурге. Технические конференции в основном проводились на западном побережье США, близ Кремниевой долины. Это доставляло неудобства жителям восточного побережья.

В этом же году Крис Нандор (Chris Nandor) написал на Perl сценарий, отправивший 25000 поддельных голосов на голосовании за звезд бейсбола в пользу шортстопера (shortstop) Номара Гарсиапарры (Nomar Garciaparra), игрока команды «Boston Red Sox»,[1] заработав ему упоминание его имени в истории голосований на несколько последующих лет. Эта история может служить примером, какое влияние может оказать короткий эпизод в телевизионном шоу «Sports Night».[2]

В следующем году лондонская группа пользователей Perl организовала конференцию YAPC::EU (которая, впрочем, была не первым крупным событием Perl в Европе; первая конференция «German Perl Workshop» предшествовала даже конференции «The Perl Conference»). Эти конференции оказались настолько успешными, что превратились в организацию «Yet Another Foundation» (также известную, как «The Perl Foundation») в США и «YAPC Europe Foundation» в Европе. Вскоре появились аналогичные конференции YAPC в Азии и Южной Америке, хотя из общего у них было только название. Теперь не проходит и недели без крупного события Perl где-нибудь в мире, что еще больше сплачивает сообщества людей, которые работают независимо друг от друга, но встречаются достаточно часто.

Конференция «The Perl Conference» продолжала расти в других направлениях. Она превратилась в конференцию «The Open Source Conference», или просто OSCON, где Ларри регулярно выступает со своей речью «State of the Onion», а Дамиан Конвей (Damian Conway) вызывает восторги аудитории своей речью «The Conway Channel». В 2000 году на конференции OSCON Ларри анонсировал Perl 6 – который не является темой этой книги – как амбициозный проект, начатый с нуля. В этой книге мы скажем лишь, что Perl 6 в значительной степени является забавой, он оживил развитие Perl 5 и помимо названия «Perl» не имеет ничего общего с языком программирования, о котором мы здесь рассказываем. Это совершенно другой язык, заимствующий кое-что из Perl, как Perl заимствует из других языков.

---

1 «Cyber-stuffing remains threat to All-Star voting» (голосование за звезд оказалось под кибернетической угрозой), ESPN.com (*http://static.espn.go.com/mlb/s/2001/0624/1218244.html*).

2 В эпизоде «Louise Revisited», вышедшем 26 октября 1999, рассказывалось, как Джереми (Jeremy) использовал сценарий на Perl для отправки поддельных голосов за Кейси (Casey), одного из телеведущих.

Продолжение истории, по крайней мере после 2002 года, см. в Perl Timeline на CPAST – Comprehensive Perl Arcana Society Tapestry (*http://history.perl.org*).

Поэзия Perl

Perl предполагает, что любое встретившееся *голове слово*, в конечном счете станет названием подпрограммы, даже если в данный момент подпрограмма не определена. Иногда такой стиль программирования называют «поэтическим». Это позволяет людям писать на языке Perl стихи вроде вот такого монстра:

```
BEFOREHAND: close door, each window & exit; wait until time
    open spellbook, study, read (scan, select, tell us);
write it, print the hex while each watches,
    reverse its length, write again;
    kill spiders, pop them, chop, split, kill them.
        unlink arms, shift, wait & listen (listening, wait),
sort the flock (then, warn the "goats" & kill the "sheep");
    kill them, dump qualms, shift moralities,
    values aside, each one;
        die sheep! die to reverse the system
        you accept (reject, respect);
next step,
    kill the next sacrifice, each sacrifice,
    wait, redo ritual until "all the spirits are pleased";
    do it ("as they say").
do it(*everyone***must***participate***in***forbidden**s*e*x*)
return last victim; package body;
    exit crypt (time, times & "half a time") & close it,
    select (quickly) & warn your next victim;
AFTERWORDS: tell nobody.
    wait, wait until time;
    wait until next year, next decade;
        sleep, sleep, die yourself,
        die at last
```

Ларри написал это стихотворение и отправил в *news.groups* в поддержку своего предложения создания группы *comp.lang.perl.poems*. Большинство, вероятно, заметило, что стихотворение было отправлено 1 апреля, но это не удержало их от создания собственных стихов на Perl.

Шэрон Хопкинс (Sharon Hopkins) написала много стихотворений на Perl, а также статью о поэзии Perl, которую представила на технической конференции Usenix зимой 1992 года, озаглавленную «Camels and Needles: Computer Poetry Meets the Perl Programming Language» (Иголки и верблюды: на стыке компьютерной поэзии и языка программирования Perl). Шэрон не только самая плодовитая из поэтов на Perl, но и самая публикуемая; следующее ее стихотворение было опубликовано в «The Economist» (*http://www.economist.com*) и «The Guardian» (*http://www.guardiannews.com*):

```
#!/usr/bin/perl

APPEAL:
```

```
listen (please, please);

open yourself, wide;
    join (you, me),
connect (us,together),

tell me.

do something if distressed;

    @dawn, dance;
    @evening, sing;
    read (books,$poems,stories) until peaceful;
    study if able;

    write me if-you-please;

sort your feelings, reset goals, seek (friends, family, anyone);

        do*not*die (like this)
        if sin abounds;

keys (hidden), open (locks, doors), tell secrets;
do not, I-beg-you, close them, yet.

                        accept (yourself, changes),
                        bind (grief, despair);

require truth, goodness if-you-will, each moment;

select (always), length(of-days)

# listen (a perl poem)
# Sharon Hopkins
# rev. June 19, 1995
```

Достоинства программиста на Perl

Лень

Лень программиста напоминает одноименный человеческий недостаток, но есть разница. Недостаток выражается в отлынивании от текущей работы. Достоинство же выражается в отлынивании от работы в будущем. Программисты, имеющие доступ к мощи Perl, создают инструменты, упрощающие решение задач. Perl – серьезный язык автоматизации задач, и чем более высокий уровень автоматизации будет достигнут сегодня, тем меньше ручного труда выпадет на долю программиста завтра.

Нетерпеливость

Нетерпеливость – жуткое чувство, возникающее, когда компьютер делает все что угодно, только не то, что нужно. Или, выражаясь точнее, когда программист по ту сторону программного продукта выбрал неправильные настройки по умолчанию, создал неудобный графический интерфейс или не предоставил

доступ к этим данным. Вы достаточно испытали этих неприятных ощущений, чтобы не заставлять других испытывать их, а, стало быть, обернуть свои расстройства и потраченное время на пользу другим.

Высокомерие

Высокомерие – это чувство, что при наличии необходимых инструментов возможным становится практически все. Решение любой сложной задачи – это Всего Лишь Вопрос Программирования, правильно? Однако это же чувство может заставить вас подлететь слишком близко к Солнцу.

События

Практически каждую неделю в экосистеме Perl происходит какое-то событие. Ниже перечислены некоторые из основных событий. Большинство из них можно найти в списке событий «The Perl Review Community Calendar» (*http://theperlreview.com/community_calendar*).

The Perl Conference, OSCON

Конференция «The Perl Conference», организованная издательством O'Reilly & Associates в 1997 году, была не первым событием в экосистеме Perl, но, вероятно, одним из важнейших. На этой конференции маленькая группа жителей Нью-Йорка организовала первую группу пользователей Perl, NY.pm (*http://ny.pm.org*). Это привело к созданию ряда других групп пользователей Perl в том же году; в течение пары лет появились еще сотни групп. «The Perl Conference» выросла до организации «The Open Source Conference», или просто OSCON.

YAPC

YAPC, или «Yet Another Perl Conference» (еще одна конференция пользователей Perl), приобрела множество форм и охватывает по меньшей мере четыре континента. Каждый год одна из этих малобюджетных, массовых и, как правило, некоммерческих конференций проводится в Азии, Европе, Северной и Южной Америке. Несмотря на одинаковые названия, это различные организации.

Perl Workshops

Конференции YAPC длятся по несколько дней, тогда как встречи «Perl Workshop» обычно занимают один-два дня и посвящены определенной теме, например, на семинаре «Perl QA Workshop» обсуждаются проблемы инфраструктуры CPAN и тестирования программ на Perl. Немногие знают, что первым событием в экосистеме Perl стала организация семинара «German Perl Workshop», прошедшего даже до созыва конференции «The Perl Conference».

Hackathons

Наименее организованным событием в экосистеме Perl являются встречи «Hackathons», на которых пользователи Perl собираются, чтобы поработать вместе. Иногда эти встречи посвящены определенной теме, а иногда люди собираются, чтобы поработать вместе над своими проектами в одной комнате.

Где и как получить помощь

Программисты на Perl чаще других стремятся прийти на помощь, это замечают даже те, кому не нравится Perl. Мы думаем, что Perl, уходящий корнями в самые разные языки программирования, привлекает людей с особым складом характера, которым нравятся разные языки, а не только тот, которым они пользуются. Возможно, они склонны находить пользу во всем.

Если вы ищете ответы на вопросы, в Интернете можно найти множество форумов, участники которых готовы прийти вам на помощь. Ниже перечислены наиболее известные из них:

http://perldoc.perl.org

Содержит всю электронную документацию по языку Perl, без которой невозможно жить и работать, даже если ваша платформа или система управления пакетами думает иначе. Да, некоторые компании поставляют *perl* без руководств.

Learn Perl (http://learn.perl.org)

Этот веб-сайт – начальная точка поиска ресурсов для начинающих, включая перечисленные здесь.

Perl beginners mailing list

Кейси Уэст (Casey West) создал этот список рассылки как надежный источник информации, где самые зеленые новички могут задавать простейшие вопросы, не опасаясь насмешек. Другие форумы могут быть более, гм-м, неуправляемыми и способны вызывать неприятные эмоции у начинающих программистов на Perl.

Perlmonks (http://www.perlmonks.org)

Perlmonks – это электронная доска объявлений, посвященная языку Perl. Конечно, это не справочное бюро, но если вы хорошо потрудились и задали интересный вопрос, вы наверняка быстро получите качественную помощь. Но для начала рекомендуем прочитать руководство брайана по решению любых проблем в Perl («brian's Guide to Solving Any Perl Problem», *http://www.perlmonks.org/?node_id=376075*).[1]

Stackoverflow (http://www.stackoverflow.com)

Stackoverflow – известный сайт, построенный по принципу «вопрос-ответ» и посвященный общим вопросам программирования. Даже при том, что он не посвящен конкретно Perl, здесь достаточно завсегдатаев, являющихся экспертами по этому языку. Они с удовольствием ответят на ваши вопросы.

Ваша местная группа пользователей Perl

В мире существуют сотни групп пользователей Perl. И хотя каждая из них имеет свои особенности, это отличный способ найти и познакомиться с пользователями Perl, живущими рядом с вами (или не рядом). Многие из этих групп проводят свои семинары и встречи. Найти ближайшую группу можно на сайте *http://www.pm.org*, а если поблизости не окажется такой группы, создайте свою!

---

1 Это руководство приводится также в книге «Mastering Perl».

Телеконференции Usenet

Телеконференции, посвященные Perl, – кладезь информации, хотя иногда и неупорядоченной. Первую остановку можно сделать на *news:comp.lang.perl.moderated*, регулируемой телеконференции с небольшой активностью. Здесь появляются обновления и проходят технические дискуссии. Благодаря регулированию, эта телеконференция вполне удобочитаема.

В группе *news:comp.lang.perl.misc* с высокой активностью обсуждается все, от технических проблем и философии Perl до игр и поэзии на Perl. Как и сам Perl, группа *news:comp.lang.perl.misc* нацелена на оказание помощи, и никакой вопрос здесь не будет выглядеть глупым.[1]

Если для доступа к Usenet вы пользуетесь браузером, а не обычным клиентом чтения новостей, добавьте префикс *news:* перед именем группы, чтобы перейти к ней. (Такой прием действует, только если у вас имеется сервер новостей.) В противном случае можно использовать службу поиска в Usenet, такую как Google Groups (*http://groups.google.com/*), указав `*perl*` в строке поиска.

Списки рассылки

Многие темы, общие или конкретные, обычно имеют собственные списки рассылки. Многие из них перечислены на сайте *http://lists.perl.org*. Списки рассылки часто можно найти на веб-сайтах проектов. Для поиска архивов многих списков рассылки можно также пользоваться такими сайтами, как *http://markmail.org*.

IRC

Чаты Интернета (Internet Relay Chat, IRC) – еще одно средство общения программистов на Perl, и, если вам нравится такой стиль общения, вы без труда найдете множество желающих поболтать с вами. Такие чаты обычно не считаются местом, где можно получить помощь, поэтому вопросы без предварительного знакомства здесь обычно расцениваются, как появление на вечеринке без приглашения. Однако некоторые каналы, такие как *#perl-help* и *#win32*, предназначены специально для оказания помощи. Кроме того, множество каналов IRC можно найти на сайте *http://www.irc.perl.org/*.

---

[1] Конечно, есть вопросы слишком глупые, чтобы отвечать на них. (Особенно если ответы на них уже имеются в электронных страницах справочного руководства и сборниках ответов на часто задаваемые вопросы. Зачем просить помощи в телеконференции, если тот же самый ответ можно найти самому гораздо раньше, чем вы закончите набирать свой вопрос?)

V

Справочный материал

25

Специальные имена

Эта глава посвящена переменным, которые имеют в Perl особое значение. Для большинства имен, содержащих знаки пунктуации, есть хорошие мнемоники или аналоги в какой-нибудь оболочке (или же и то и другое). Но если вы хотите использовать в качестве синонимов длинные имена переменных, просто скажите в начале программы:

```
use English "-no_match_vars";
```

В результате в текущем пакете для всех коротких имен будут установлены псевдонимы в виде длинных имен. У некоторых из этих переменных даже есть средние имена, обычно заимствованные из *awk*. Большинство в конечном счете останавливает выбор на коротких именах, по крайней мере, для самых востребованных переменных. В данной книге мы систематически используем короткие имена, но часто указываем и длинные (в скобках), чтобы их легко можно было найти в данной главе.

Семантика этих переменных бывает очевидно волшебной. (Как творить собственное волшебство, мы рассказали в главе 14.) Некоторые из них доступны только для чтения. При попытке присвоить им значения возникает исключительная ситуация.

Ниже сначала приводится полный список переменных и функций, имеющих в Perl особое значение и сгруппированных по типу, чтобы можно было находить переменные, в точном названии которых вы не уверены. Затем следуют описания всех переменных в алфавитном порядке их правильных имен (или наименее неправильных).

Специальные имена, сгруппированные по типам

Мы вольно употребили слово «тип» – представленные разделы объединяют переменные скорее по области видимости.

Специальные переменные регулярных выражений

Следующие специальные переменные, связанные с поиском по шаблону, видны во всей динамической области видимости, где выполняется поиск. Иными словами, ведут себя, как будто объявлены как `local`**, поэтому вам не нужно самим объявлять их таким образом. См. главу 5.**

```
$digits

$& ($MATCH)
$' ($POSTMATCH)
$` ($PREMATCH)

${^MATCH}
${^POSTMATCH}
${^PREMATCH}

$+ ($LAST_PAREN_MATCH)
%+ (%LAST_PAREN_MATCH)
@+ (@LAST_MATCH_END)

@-
%-

$^R ($LAST_REGEXP_CODE_RESULT)
$^N ($LAST_SUBMATCH_RESULT)
```

Переменные дескриптора файла

Эти специальные переменные не требуется определять как `local`**, потому что они всегда указывают на значение, относящееся к выбранному в данный момент дескриптору вывода, – каждый дескриптор сохраняет собственный набор значений. Когда с помощью** `select` **выбирается новый дескриптор файла, прежний запоминает значения, которые имели эти переменные, а переменные начинают отражать значения нового дескриптора. См. также описание модуля** `IO::Handle`**.**

```
$| ($AUTOFLUSH, $OUTPUT_AUTOFLUSH)
$- ($FORMAT_LINES_LEFT)
$= ($FORMAT_LINES_PER_PAGE)
$~ ($FORMAT_NAME)
$% ($FORMAT_PAGE_NUMBER)
$^ ($FORMAT_TOP_NAME)
```

Специальные переменные пакетов

Эти переменные существуют отдельно в каждом пакете. Нужды в их локализации не должно быть, поскольку `sort` **автоматически делает это для** `$a` **и** `$b`**, а остальные, скорее всего, лучше оставить в покое (хотя при использовании директивы** `use strict` **потребуется объявить их как** `our`**).**

```
$a
$AUTOLOAD
$b
@EXPORT
```

```
@EXPORT_OK
%EXPORT_TAGS
%FIELDS
@ISA
%OVERLOAD
$VERSION
```

Специальные переменные для всей программы

Эти переменные действительно являются глобальными в самом широком смысле – они означают одно и то же в каждом пакете, так как переадресовываются в пакет `main`, при использовании неквалифицированных имен (за исключением `@F`, которая является специальной в `main`, но принудительная переадресация на нее не выполняется). Если потребуется временный экземпляр одной из этих переменных, локализуйте его в текущей динамической области видимости.

```
%ENV
%! (%ERRNO, %OS_ERROR)
%INC
%SIG
%^H

@_
@ARGV
@INC

$_
$0 ($PROGRAM_NAME)
$ARGV

$! ($ERRNO, $OS_ERROR)
$" ($LIST_SEPARATOR)
$$ ($PID, $PROCESS_ID)
$( ($GID, $REAL_GROUP_ID)
$) ($EGID, $EFFECTIVE_GROUP_ID)
$  ($OFS, $OUTPUT_FIELD_SEPARATOR)
$  ($NR, $INPUT_LINE_NUMBER)
$/ ($RS, $INPUT_RECORD_SEPARATOR)
$. ($FORMAT_LINE_BREAK_CHARACTERS)
$; ($SUBSEP, $SUBSCRIPT_SEPARATOR)
$< ($UID, $REAL_USER_ID)
$> ($EUID, $EFFECTIVE_USER_ID)
$? ($CHILD_ERROR)
$@ ($EVAL_ERROR)
$[
$\ ($ORS, $OUTPUT_RECORD_SEPARATOR)
$]
$^A ($ACCUMULATOR)
$^C ($COMPILING)
$^D ($DEBUGGING)
${^ENCODING}
$^E ($EXTENDED_OS_ERROR)
${^GLOBAL_PHASE}
$^F ($SYSTEM_FD_MAX)
```

```
$^H
$^I ($INPLACE_EDIT)
$^L ($FORMAT_FORMFEED)
$^M
$^O ($OSNAME)
${^OPEN}
$^P ($PERLDB)
$^R ($LAST_REGEXP_CODE_RESULT)
${^RE_DEBUG_FLAGS}
${^RE_TRIE_MAXBUF}
$^S ($EXCEPTIONS_BEING_CAUGHT)
$^T ($BASETIME)
${^TAINT}
${^UNICODE}
${^UTF8CACHE}
${^UTF8LOCALE}
$^V ($PERL_VERSION)
$^W ($WARNING)
${^WARNING_BITS}
${^WIDE_SYSTEM_CALLS}
${^WIN32_SLOPPY_STAT}
$^X ($EXECUTABLE_NAME)
```

Специальные дескрипторы файлов для пакетов

За исключением дескриптора DATA, который всегда относится к текущему пакету, следующие дескрипторы файлов относятся к main, если не квалифицированы именем другого пакета:

```
_ # (подчеркивание)
ARGV
ARGVOUT
DATA
STDIN
STDOUT
STDERR
```

Специальные функции для пакетов

Следующие имена подпрограмм имеют для Perl особое значение. Они всегда вызываются неявно при возникновении некоторых событий, таких как доступ к связанным переменным или попытка вызвать не определенную функцию. Мы не описываем их в этой главе, поскольку они широко освещаются на протяжении всей книги.

Перехватчик вызова неопределенной функции (см. главу 10):

```
AUTOLOAD
```

Завершение отживших свое объектов (см. главу 12):

```
DESTROY
```

Объекты исключительных ситуаций (см. die в главе 27):

```
PROPAGATE
```

Функции автоматической инициализации и автоматической уборки (см. главу 18):

```
BEGIN, CHECK, UNITCHECK, INIT, END
```

Поддержка многопоточной модели выполнения

```
CLONE, CLONE_SKIP
```

Методы связывания (см. главу 14):

```
BINMODE, CLEAR, CLOSE, DELETE, DESTROY, EOF, EXISTS, EXTEND,
FETCH, FETCHSIZE, FILENO, FIRSTKEY, GETC, NEXTKEY, OPEN, POP,
PRINT, PRINTF, PUSH, READ, READLINE, SCALAR, SEEK, SHIFT,
SPLICE, STORE, STORESIZE, TELL, TIEARRAY, TIEHANDLE, TIEHASH,
TIESCALAR, UNSHIFT, WRITE.
```

Специальные переменные в алфавитном порядке

Мы привели эти переменные в алфавитном порядке их длинных имен. Если вы не знаете длинного имени переменной, можете найти его в предыдущем разделе. (Переменные, не имеющие буквенного имени, вынесены вперед.)

Чтобы не повторяться, каждое описание переменной начинается с одного из обозначений, перечисленных в табл. 25.1:

Таблица 25.1. Обозначения для специальных переменных

Обозначение	Смысл
XXX	Устарела, *не используйте* в новом коде.
NOT	Not Officially There (только для внутреннего использования).
RMV	Удалена из Perl.
ALL	Подлинно глобальная, используется всеми пакетами.
PKG	Глобальная для пакета; в каждом пакете может быть своя.
FHA	Атрибут дескриптора файла; один на каждый объект ввода/вывода.
DYN	Автоматическая динамическая область видимости (подразумевается ALL).
LEX	Лексическая область видимости на этапе компиляции.
RO	Только для чтения (Read Only), создает исключительную ситуацию при модификации.

Если перечислено несколько имен переменных или символов, по умолчанию используется только короткое имя. Применение модуля English делает доступными длинные синонимы и только в текущем пакете, даже если переменная помечена как [ALL].

Заголовки статей, имеющие вид *method HANDLE EXPR*, отражают объектно-ориентированные интерфейсы к переменным для дескрипторов файлов, предоставляемым модулем IO::Handle и различными модулями IO::. (При желании можно использовать форму записи *HANDLE->method (EXPR)*.) Они позволяют избежать необходимости вызывать select для смены дескриптора файла вывода по умолчанию

при просмотре или изменении этой переменной. Все такие методы возвращают прежнее значение атрибута; новое значение присваивается, если задан аргумент *EXPR*. Если он не задан, большинство методов ничего не делает с текущим значением, за исключением autoflush, который принимает аргумент 1, просто чтобы чем-то выделиться.

$_ [ALL] Пространство по умолчанию для ввода и поиска по шаблону. Следующие пары эквивалентны:

```
while (<>) {...} # эквивалентны только в обычной проверке while
while (defined($_ = <>)) {...}

chomp
chomp($_)

/^Subject:/
$_ =~ /^Subject:/

tr/a-z/A-Z/
$_ =~ tr/a-z/A-Z/
```

Ниже перечислено, где по умолчанию используется переменная $_, если явно не определено другое:

- Списочные функции, например print, unlink; унарные функции, такие как ord, pos и int; а также все проверки файлов, кроме -t, которая по умолчанию выполняется для STDIN. Все функции, по умолчанию использующие $_, отмечены также в главе 27.
- Операции поиска по шаблону m// и s///, а также операции транслитерации y/// и tr///, когда они используются без оператора =~.
- Переменная итерации в цикле foreach (даже в виде for или когда выступает в качестве модификатора команды), если не задана другая переменная.
- Неявная переменная итерации в функциях grep и map. (Задать для них другую переменную невозможно.)
- Место по умолчанию для входной записи, когда истинность результата операции <FH>, readline или glob является единственным критерием для оператора while. Это присваивание не производится вне выражения проверки условия while или когда в это выражение включены дополнительные элементы.

Поскольку $_ является глобальной переменной, она иногда может давать нежелательные побочные эффекты. Начиная с версии v5.10 имеется возможность использовать приватную (лексическую) версию переменной $_, объявив ее как my. Кроме того, объявление our $_ восстанавливает доступ к глобальной переменной $_ в текущей области видимости.

(Мнемоника: подчеркивание в некоторых операциях подчеркивает используемый операнд.)

@_ [ALL] Внутри подпрограммы этот массив содержит список аргументов, переданных подпрограмме. См. главу 7.

_ (подчеркивание)

[ALL] Специальный дескриптор файла, используемый для кэширования информации последнего успешного выполнения оператора `stat`, `lstat` или проверки файла (такой как `-w $file` или `-d $file`).

$цифры

[DYN,RO] Нумерованные переменные `$1`, `$2` и т. д. (до нужной вам величины)[1] содержат текст, соответствующий очередной паре круглых скобок последнего успешно сопоставленного шаблона в активной в данный момент динамической области видимости. (Мнемоника: как `\цифры`.)

$] [ALL] Возвращает номер версии + уровень исправлений (patchlevel/1000). Может использоваться в начале сценария для определения пригодности имеющейся версии интерпретатора Perl. (Мнемоника: эта версия Perl в правильных рамках?) Пример:

```
warn "No checksumming!\n" if $] < 3.019;
die "Must have prototyping available\n" if $] < 5.003;
```

См. также описание директив `use VERSION` и `require VERSION`, обеспечивающих более удобный способ завершения работы, если версия интерпретатора Perl слишком старая. См. описание `$^V` как более гибкого представления версии Perl.

$[[XXX,LEX] Индекс первого элемента в массиве и первого символа в подстроке. По умолчанию 0, но мы, бывало, устанавливали его в 1, чтобы Perl больше походил на *awk* (или FORTRAN) при индексации и вычислении функций `index` и `substr`. Поскольку присваивание `$[` было признано очень опасным, оно интерпретируется теперь как директива компилятора с лексической видимостью и не может повлиять на какие-либо другие файлы. (Мнемоника: `[` начинает индексы.)

$# [RMV,ALL] Удалена в версии v5.10. Не применяйте ее, используйте вместо нее `printf`. `$#` содержит формат вывода чисел через `print`, что было нерешительной попыткой эмулировать переменную `OFMT` из *awk*. (Мнемоника: `#` является знаком номера, но если вы умны (sharp)[2], то забудьте об этом, чтобы не превратить свою программу в мешанину и понести за это наказание.)

Это не разыменовывающий символ, который используется перед именами массивов, чтобы определить индекс последнего элемента, как в выражении `$#ARRAY`. Эти две пары символов никак не связаны друг с другом.

$* [RMV,ALL] Эту ныне усопшую переменную когда-то можно было установить в истинное значение, чтобы заставить Perl предполагать наличие модификатора `/m` в каждом шаблоне, где отсутствует явный модификатор `/s`. Была удалена в Perl версии v5.10. (Мнемоника: `*` соответствует множеству вещей.)

---

[1] Хотя многие механизмы регулярных выражений поддерживают лишь до девяти обратных ссылок на найденный текст, в Perl такого ограничения нет, поэтому если написать `$768`, Perl не станет возражать, в отличие от тех, кому придется сопровождать этот код, если в регулярном выражении действительно использовано столько скобок.

[2] Игра слов – «sharp» в переводе также означает «диез» (знак #). – *Прим. перев.*

%- **[DYN,RO] Действует подобно** %+ (%LAST_PAREN_MATCH). **Обеспечивает доступ к именованным сохраняющим группам в последней успешной операции поиска по шаблону, в текущей активной динамической области видимости. Ключами этого хеша являются имена сохраняющих групп, а значениями – массив ссылок. Каждый массив содержит фрагменты, соответствующие всем группам с тем же именем, которых может быть несколько, в порядке следования в шаблоне.**

Не смешивайте вызовы each **с этим хешем и поиск по шаблону в цикле, иначе вы будете получать противоречивые результаты.**

Если вам не нравится форма записи $-{NAME}[0], **используйте стандартный модуль** Tie::Hash::NamedCapture, **чтобы создать собственный псевдоним для переменной** %-.

$a **[PKG] Эта переменная используется функцией** sort **для хранения первого из пары сравниваемых значений** ($b **содержит второе значение каждой пары). Переменная** $a **принадлежит пакету, куда был скомпилирован оператор** sort, **и не обязательно тому же, куда была скомпилирована его функция сравнения. Эта переменная неявно локализуется в блоке сравнения** sort **и является глобальной, так что не вызывает претензий со стороны** use strict. **Поскольку это псевдоним для фактического массива, может показаться, что массив можно модифицировать через переменную, однако делать этого не стоит. См. описание** sort.

$ACCUMULATOR

$^A **[ALL] Текущее значение накопителя** write **для строк** format. **Формат содержит команды** formline, **которые помещают свой результат в** $^A. **После вызова своего формата** write **выводит содержимое** $^A **и очищает ее. Поэтому вы никогда фактически не видите содержимого** $^A, **если только не вызовете** formline **самостоятельно, а потом не посмотрите на результат вызова. См. описание функции** formline.

ARGV

[ALL] Специальный дескриптор файла, который перебирает все содержащиеся в командной строке имена файлов (из переменной @ARGV). **Обычно записывается как пустой дескриптор файла в операторе угловых скобок: <>.**

$ARGV

[ALL] Содержит имя текущего файла при чтении из дескриптора ARGV **в случае применения операторов <> или** readline.

@ARGV

[ALL] Массив с аргументами командной строки для сценария. Заметьте, что значение $#ARGV **обычно равно числу аргументов минус один, так как** $ARGV[0] **является первым аргументом, а не именем команды; используйте** scalar @ARGV **для получения количества аргументов программы. Имя программы можно найти в** $0.

ARGVOUT

[ALL] Специальный указатель файла, используемый при обработке указателя ARGV **с ключом** ***-i*** **или переменной** $^I. **См. описание ключа** ***-i*** **в главе 17.**

$AUTOLOAD

[PKG] В ходе выполнения метода AUTOLOAD эта переменная, глобальная для пакета, содержит полное квалифицированное имя функции, от имени которой был запущен метод AUTOLOAD. См. главу 25.

$b [PKG] Эта переменная, спутница $a, используется в сравнениях sort. Подробности см. в описании $a и функции sort.

$BASETIME

$^T [ALL] Момент времени, когда начал выполняться сценарий, в секундах после начала эпохи (для систем UNIX – начало 1970 года). Значения, возвращаемые проверками файлов -M, -A и -C, вычисляются относительно этого момента времени.

$CHILD_ERROR

$? [ALL] Код состояния, полученный последней операцией закрытия канала, командой `` (обратные апострофы) или функциями wait, waitpid и system. Обратите внимание, что это не просто код завершения, а целое 16-разрядное слово состояния, возвращаемое системными вызовами *wait*(2) или *waitpid*(2) (или эквивалентными). Код завершения порожденного процесса находится в старшем байте, т.е. $? >> 8. Результат операции с младшим байтом $? & 127 сообщает, какой сигнал (если он был) послужил причиной завершения процесса, а результат операции $? & 128 сообщает, последовала ли за его завершением операция записи образа памяти на диск. (Мнемоника: аналогична $? в *sh* и ее потомках.)

Внутри блока END переменная $? содержит значение, которое должно быть передано exit. Переменную $? можно изменить в END, чтобы вернуть другой код завершения сценария. Например:

```
END {
    $? = 1 if $? == 255; # теперь die вернет код завершения 255
}
```

В системе VMS использование прагмы use vmsish "status" приводит к тому, что в $? отражается подлинный код завершения VMS, а не эмулируемый по умолчанию код завершения POSIX.

Если переменная h_errno поддерживается в C, ее числовое значение возвращается через $?, когда одна из функций gethost*() терпит неудачу.

$COMPILING

$^C [ALL] Текущее состояние внутреннего флага, связанное с ключом -c. В основном полезно в сочетании с *-MO=...*, чтобы позволить коду изменять свое поведение. Например, метод AUTOLOAD может потребоваться выполнить на этапе компиляции, вместо использования обычной отложенной загрузки, чтобы код был сгенерирован немедленно. Установка значения $^C = 1 сродни вызову B::minus_c. См. главу 16.

DATA

[PKG] Этот специальный дескриптор файла ссылается на все, что расположено в текущем файле за маркерами __END__ или __DATA__. Маркер __END__ всегда открывает дескриптор файла main::DATA, поэтому используется в основной программе. Маркер __DATA__ открывает дескриптор DATA в текущем пакете,

поэтому различные модули могут иметь собственные дескрипторы файла DATA, поскольку они (предположительно) имеют различные имена пакетов.

$DEBUGGING

$^D [ALL] Текущее значение внутренних флагов отладки, установленных ключом командной строки *-D*; значения разрядов см. в разделе «Ключи», в главе 17. По аналогии с командной строкой, этой переменной можно присваивать числовые или символьные значения, например: $^D = 10 или $^D = "st".

(Мнемоника: значение ключа *-D*.)

${^ENCODING}

[XXX,ALL] *Ссылка на объект* Encode, используемый для преобразования исходного кода в Юникод. Благодаря этой переменной отпадает необходимость писать сценарии на Perl в кодировке UTF-8. По умолчанию имеет значение undef. Прямое управление этой переменной не рекомендуется.

Была добавлена в Perl версии v5.8.2.

$EFFECTIVE_GROUP_ID

$) [ALL] Текущий GID (ID группы) данного процесса. Если платформа поддерживает одновременное членство в нескольких группах, $) позволяет получить список групп, элементы которого разделены пробелами. Первое число равно значению, возвращаемому *getegid*(2), а последующие – числам, возвращаемым *getgroups*(2), одно из которых может совпадать с первым.

Точно так же значение, присваиваемое $), должно быть списком чисел, элементы которого разделены пробелами. Первое число используется для установки текущего GID, а остальные (если имеются) передаются системному вызову *setgroups*(2). Чтобы задать для setgroups пустой список, просто повторите новый текущий GID; например, чтобы установить текущий GID в значение 5 и указать пустой список для setgroups, скажите:

```
$) = "5 5";
```

(Мнемоника: круглые скобки применяются для *группировки* объектов. Текущий GID – это *ваша группа*, если вы выполняете setgid.) Заметьте: $<, $>, $(и $) могут устанавливаться только на машинах, поддерживающих соответствующую системную процедуру set-id. $(и $) можно менять местами только на машинах, поддерживающих *setregid*(2).

$EFFECTIVE_USER_ID

$> [ALL] Текущий UID данного процесса, возвращаемый системным вызовом *geteuid*(2). Пример:

```
$< = $>;            # присвоить действительный uid текущему
($<,$>) = ($>,$<); # поменять местами действительный и текущий uid
```

Текущий и действительный UID можно изменить одновременно с помощью POSIX::setuid. После изменения $> необходимо проверить $!, чтобы выявить возможные ошибки, возникшие при попытке изменения.

(Мнемоника: это UID, *на который* вы перешли при выполнении setuid.) Примечание: $< и $> можно менять местами только на машинах, поддерживающих *setreuid*(2). И то не всегда.

%ENV

[ALL] Хеш, содержащий текущие переменные среды. Присваивание %ENV изменяет среду как главного процесса, так и процессов, порожденных главным после присваивания. (Таким способом нельзя изменить окружение родительского процесса в любой системе, сходной с UNIX.)

```
$ENV{PATH}  = "/bin:/usr/bin";
$ENV{PAGER} = "less";
$ENV{LESS}  = "MQeicsnf"; # наши любимые ключи для less(1)
system "man perl";        # получает новые установки
```

Чтобы удалить что-либо из окружения, вызовите функцию delete вместо присваивания значения undef элементу хеша.

Обратите внимание, что процессы, выполняющиеся как записи *crontab*(5), наследуют особенно узкий набор переменных среды. (Если программа прекрасно выполняется из командной строки, но не под *cron*, возможно, причина в этом.) Заметьте также, что необходимо установить $ENV{PATH}, $ENV{SHELL}, $ENV{BASH_ENV} и $ENV{IFS}, если сценарий выполняется как setuid. См. главу 20.

$EVAL_ERROR

$@ [ALL] Текущее активное исключение или сообщение о синтаксической ошибке Perl в последней операции eval. (Мнемоника: в каком месте («at») была синтаксическая ошибка?) В отличие от $! ($OS_ERROR), которая устанавливается при неудаче, но не очищается при успехе, $@ всегда устанавливается (в истинное значение), если во время выполнения в последнем вызове eval возникла ошибка компиляции или исключительная ситуация, и всегда сбрасывается (в ложное значение), если таких проблем не возникло.

Предупреждения в $@ не сохраняются. Однако можно определить процедуру обработки предупреждений, установив $SIG{__WARN__}, как описывается далее в этом разделе.

Заметьте, что значением $@ может быть не строка, а объект исключительной ситуации. Даже в этом случае с ним, вероятно, можно работать как со строкой, если в классе объекта исключительной ситуации определена перегруженная операция преобразования в строку. Если для распространения исключительной ситуации применяется команда:

```
die if $@;
```

объект исключительной ситуации вызовет $@->PROPAGATE, чтобы решить, что делать. (Строковая исключительная ситуация просто добавляет к строке «propagated at».)

$EXCEPTIONS_BEING_CAUGHT

$^S [ALL] Эта переменная отражает текущее состояние интерпретатора, возвращая истинное значение при нахождении внутри eval и ложное в противном случае. Она не определена, если анализ текущей единицы компиляции еще не завершен, что может происходить в обработчиках $SIG{__DIE__} и $SIG{__WARN__}. (Мнемоника: состояние eval.)

$EXECUTABLE_NAME

$^X [ALL] Название двоичного модуля *perl*, взятое из argv[0] в C.

@EXPORT

[PKG] К этой переменной-массиву обращается метод import модуля Exporter при поиске списка других переменных и подпрограмм пакета, которые нужно экспортировать по умолчанию при загрузке модуля директивой use или в случае применения тега импорта :DEFAULT. Эта переменная не является исключением для use strict, поэтому ее нужно объявлять как our или использовать полное имя, квалифицированное именем пакета, в области действия этой прагмы. Однако все переменные, имена которых начинаются строкой «EXPORT», исключаются из предупреждений об однократном применении. См. главу 11.

@EXPORT_OK

[PKG] К этой переменной-массиву обращается метод import модуля Exporter, чтобы определить, является ли законным запрашиваемый импорт. Она не является исключением для use strict. См. главу 11.

%EXPORT_TAGS

[PKG] К этой переменной-хешу обращается метод import модуля Exporter, когда запрашивается символ импорта с ведущим двоеточием, как в use POSIX ":sys_wait_h". Ключами являются теги с двоеточиями без ведущего двоеточия. Значения должны быть ссылками на массивы, содержащие символы, которые следует импортировать, когда запрашивается тег с двоеточием, и все эти значения должны также присутствовать в @EXPORT или @EXPORT_OK. Переменная не является исключением для use strict. См. главу 11.

$EXTENDED_OS_ERROR

$^E [ALL] Информация об ошибке, специфичная для используемой операционной системы. В UNIX переменная $^E идентична $! ($OS_ERROR), но это не так в OS/2, VMS, системах Microsoft и в MacPerl. Специфическую информацию можно найти в документации по конкретной версии Perl для той или иной платформы. Предостережения в описании $! обычно относятся и к $^E. (Мнемоника: дополнительное толкование ошибок.)

@F [PKG] Массив полей расщепленной строки ввода, если задан ключ командной строки *-a*. Если ключ *-a* отсутствует, массив не имеет специального значения (и фактически представляет собой лишь @main::F, а не массив во всех пакетах).

%FIELDS

[XXX,PKG] Этот хеш предназначен для внутреннего использования в прагме use fields для определения текущих допустимых полей в хеше объекта.

format_formfeed *HANDLE EXPR*

$FORMAT_FORMFEED

$^L [ALL] То, что неявно выводит функция write для подачи листа перед выводом заголовка формы. Значение по умолчанию равно "\f".

format_lines_left *HANDLE EXPR*

[FHA] Число строк, оставшихся на странице выбранного в данный момент дескриптора файла вывода, для использования в объявлении format и функции write. (Мнемоника: строк_на_странице - строк_напечатано.)

format_lines_per_page *HANDLE EXPR*

$FORMAT_LINES_PER_PAGE

$= **[FHA] Текущая длина страницы (количество строк для печати), соответствующая выбранному в данный момент дескриптору файла вывода, для использования в** format **и функции** write**. По умолчанию равна 60. (Мнемоника: символ «=» состоит из горизонтальных линий.)**

format_line_break_characters *HANDLE EXPR*

$FORMAT_LINE_BREAK_CHARACTERS

$: **[ALL] Текущий набор символов, после которых строка может быть разбита для заполнения полей продолжения (начинающихся с** ^**) в формате. Значение по умолчанию** " \n-"**, что означает разбиение по пробельным символам и дефисам. (Мнемоника: colon[1] (двоеточие) является техническим термином, означающим в поэзии часть строчки. Теперь нужно только запомнить мнемонику...)**

format_name *HANDLE EXPR*

$FORMAT_NAME

$~ **[FHA] Имя текущего формата отчета для текущего дескриптора выходного файла. Значением по умолчанию является имя дескриптора файла. (Мнемоника: поворот после** $^**.)**

format_page_number *HANDLE EXPR*

$FORMAT_PAGE_NUMBER

$% **[FHA] Номер текущей страницы для текущего дескриптора выходного файла, для использования с** format **и** write**. (Мнемоника:** % **служит регистром номера страницы в *troff*(1). Как, вы не знаете, что такое *troff*?)**

format_top_name *HANDLE EXPR*

$FORMAT_TOP_NAME

$^ **[FHA] Имя текущего формата верхнего колонтитула для текущего дескриптора выходного файла. По умолчанию представляет имя дескриптора файла с окончанием** _TOP**. (Мнемоника: указывает на верх страницы.)**

$^H **[NOT,LEX] Эта переменная содержит разряды статуса для компилятора Perl (известные также как «подсказки», hints), имеющие лексическую область видимости. Эта переменная предназначена исключительно для внутреннего использования. Ее наличие, действие и содержимое могут изменяться без уведомления. Тот, кто до нее дотронется, несомненно, умрет ужасной смертью от какой-нибудь отвратительной тропической болезни, неизвестной науке. (Мнемоника: знаем, но не скажем.)**

%^H **[NOT,LEX] Хеш** %^H **обеспечивает такую же семантику лексической видимости, как и** $^H**, что делает его полезным для реализации прагм с лексической областью видимости. Прочтите зловещие предупреждения в описании** $^H **и добавьте к ним тот факт, что данная переменная все еще является экспериментальной.**

[1] Колон (греч. kolon) – ритмическая единица прозаической речи. – *Прим. ред.*

%INC

[ALL] Хеш с именами всех файлов, загруженных посредством do *FILE*, require **или** use. **Ключами служат имена файлов, а значениями – фактические местонахождения файлов. Оператор** require **использует этот массив, чтобы определить, не загружен ли уже указанный файл. Например:**

```
% perl -MLWP::Simple -le 'print $INC{"LWP/Simple.pm"}'
/opt/perl/5.6.0/lib/site_perl/LWP/Simple.pm
```

@INC

[ALL] Массив, содержащий список каталогов, где do *FILE*, require **или** use **могут искать модули Perl. Первоначально состоит из аргументов ключа командной строки -*I* и каталогов в переменной среды** PERL5LIB, **за которыми следуют библиотеки Perl по умолчанию, например:**

```
/usr/local/lib/perl5/site_perl/5.14.2/darwin-2level
/usr/local/lib/perl5/site_perl/5.14.2
/usr/local/lib/perl5/5.14.2/darwin-2level
/usr/local/lib/perl5/5.14.2
/usr/local/lib/perl5/site_perl
.
```

за которыми следует символ «.», представляющий текущий каталог. Чтобы изменить этот список в программе, включите прагму use lib, **которая не только модифицирует переменную на этапе компиляции, но также добавляет некоторые каталоги, специфические для платформы (например, содержащие библиотеки совместного доступа, используемые модулями XS):**

```
use lib "/mypath/libdir/";
use SomeMod;
```

$INPLACE_EDIT

$^I **[ALL] Текущее значение расширения для редактирования по месту. Отключить редактирование по месту можно, назначив переменной значение** undef. **К этой переменной можно обратиться из программы для получения такого же результата, как при использовании ключа -*i*. Например, чтобы получить эквивалент команды**

```
% perl -i.orig -pe 's/foo/bar/g' *.c
```

можно выполнить в программе такой код:

```
local $^I   = ".orig";
local @ARGV = glob("*.c");
while (<>) {
    s/foo/bar/g;
    print;
}
```

(Мнемоника: значение ключа -*i*.)

$INPUT_LINE_NUMBER

$. **[ALL] Номер текущей записи (обычно номер строки) последнего дескриптора файла, из которого производилось чтение (или для которого вызывалась функция** seek **или** tell**). Значение может отличаться от фактического физического номера строки в файле – в зависимости от того, как определено текущее**

понятие «строки» (см. описание $/ ($INPUT_RECORD_SEPARATOR)). Явное закрытие дескриптора файла сбрасывает номер строки. Поскольку <> не выполняет закрытие явно, строки имеют сквозную нумерацию в файлах ARGV (но см. примеры для eof). Локализация $. локализует и представление Perl о «последнем дескрипторе файла, из которого производилось чтение». (Мнемоника: во многих программах «.» используется в качестве номера текущей строки.)

$INPUT_RECORD_SEPARATOR

$/ [ALL] Разделитель входных записей, по умолчанию – символ перевода строки, который применяется функциями readline, оператором <FH> и функцией chomp. Действует подобно переменной RS в *awk* и при установке в нулевую строку рассматривает одну или более пустых строк как признак конца записи. (Пустая строка при этом не должна содержать пробелы или табуляции.) Переменной можно присвоить строку из нескольких символов, и тогда разделение будет выполняться по многосимвольному разделителю, но нельзя присвоить шаблон – должен же *awk* быть в чем-то лучше.

Заметьте, что установка $/ в "\n\n" имеет смысл, несколько отличающийся от установки в "", если файл содержит пустые строки, располагающиеся подряд. При установке этой переменной в значение "" две или более последовательные пустые строки рассматриваются как одна пустая строка. Установка в "\n\n" означает, что Perl будет рассматривать третий перевод строки как принадлежащий новому абзацу.

Если сделать $/ неопределенной, очередная операция ввода строки «проглотит» остаток файла как одну скалярную величину:

```
undef $/;          # включить режим чтения файла целиком
$_ = <FH>;         # теперь весь файл в переменной
s/\n[ \t]+/ /g;    # выполнить свертку отступов в строках
```

Если конструкция while (<>) применяется для доступа к дескриптору ARGV при неопределенном значении $/, каждая операция чтения будет получать очередной файл:

```
undef $/;
while (<>) { # $_ содержит очередной файл целиком
    ...
}
```

Выше была использована undef, но безопаснее сделать $/ неопределенной с помощью local:

```
{
    local $/;
    $_ = <FH>;
}
```

Присваивание переменной $/ ссылки на целое число, скаляр, содержащий целое число, или скаляр, который можно преобразовать в целое число, заставит операции readline и <FH> читать записи фиксированной длины (при этом максимальным размером записи будет это целое число) вместо записей переменной длины, оканчивающихся указанной строкой. Поэтому код:

```
$/ = \32768; # или \"32768" или \$scalar_var_containing_32768
open(FILE, $myfile);
```

```
$record = <FILE>,
```

прочитает запись не длиннее 32768 байт из дескриптора *FILE*. Если чтение выполняется не из файла, ориентированного на записи (или операционная система не поддерживает файлы, ориентированные на записи), то, вероятно, при каждом чтении вы будете получать файл целиком. Если запись длиннее установленного размера, вы получите запись частями. Режим записей можно смешивать с построчным режимом только в системах, где стандартный ввод/вывод предоставляет функцию *read*(3); VMS является известным исключением.

Вызов chomp, **когда** $/ **активизирует режим записей или не определена, не оказывает эффекта. См. также ключи командной строки *-0* (цифра) и *-l* (буква) в главе 17. (Мнемоника: / используется в качестве разделителя строк при цитировании стихотворений.)**

@ISA

[PKG] Этот массив содержит имена других пакетов, которые нужно просматривать, если вызываемый метод отсутствует в текущем пакете. Это значит, что он содержит базовые классы пакета. Неявно устанавливается директивой base. **Не исключается из сообщений** strict. **См. главу 12.**

@LAST_MATCH_END

@+ **[DYN,RO] Данный массив хранит смещения конечных координат последних успешных вложенных соответствий в активной в данный момент динамической области видимости.** $+[0] **представляет смещение конца всего соответствия. Это же значение возвращает функция** pos **для переменной, для которой проводилось сопоставление. (Говоря «смещение конца», мы в действительности имеем в виду смещение первого символа, *следующего за концом* найденного соответствия, что позволяет вычислить смещения начал из смещений концов и получить длину.) *n*-й элемент этого массива содержит смещение *n*-го вложенного соответствия, поэтому** $+[1] **является смещением конца для** $1, $+[2] **– смещением конца для** $2 **и т. д. С помощью** $#+ **можно определить количество подгрупп в последнем успешном сопоставлении. См. также** @- (@LAST_MATCH_START).

После успешного поиска в некоторой переменной $var:

- $` **то же самое, что** substr($var, 0, $-[0])
- $& **то же самое, что** substr($var, $-[0], $+[0] - $-[0])
- $' **то же самое, что** substr($var, $+[0])
- $1 **то же самое, что** substr($var, $-[1], $+[1] - $-[1])
- $2 **то же самое, что** substr($var, $-[2], $+[2] - $-[2])
- $3 **то же самое, что** substr($var, $-[3], $+[3] - $-[3]) **и т. д.**

@LAST_MATCH_START

@- **[DYN,RO] Данный массив содержит смещения начал последних успешных вложенных соответствий в активной в данный момент динамической области видимости.** $-[0] **является смещением начала всего соответствия. *n*-й элемент этого массива содержит смещение *n*-го вложенного соответствия, поэтому** $-[1] **равно смещению начала для** $1, $-[2] **– смещению начала для** $2, **и т. д. С помощью** $#- **можно определить количество подгрупп в последнем успешном сопоставлении. См. также** @+ (@LAST_MATCH_END).

$LAST_PAREN_MATCH

$+ **[DYN,RO]** Эта переменная возвращает последнее вложенное соответствие в скобках для последнего найденного шаблона в текущей динамической области видимости. Это полезно, когда неизвестно (или безразлично), с каким шаблоном из группы альтернативных шаблонов было найдено соответствие. (Мнемоника: будь решительным и смотри вперед.) Пример:

```
$rev = $+ if /Version: (.*)|Revision: (.*)/;
```

%LAST_PAREN_MATCH

%+ **[DYN,RO]** Как и %-, эта переменная обеспечивает доступ к именованным сохраняющим группам в последнем успешном поиске по шаблону в текущей активной динамической области видимости. Ключами этого хеша служат имена сохраняющих групп, а значениями – строки соответствий с группами или, если есть несколько групп с одинаковым именем, строка соответствия последней группе. Если требуется получить соответствия всем одноименным группам, используйте %-.

Не смешивайте вызовы each с этим хешем и поиск по шаблону в цикле, иначе вы будете получать противоречивые результаты.

Если вам не нравится форма записи $+{*NAME*}, используйте стандартный модуль Tie::Hash::NamedCapture, чтобы создать свой псевдоним для переменной %+.

$LAST_REGEXP_CODE_RESULT

$^R **[DYN]** Эта переменная содержит результат последнего выполнения фрагмента кода в конструкции (?{ *CODE* }) в ходе успешного поиска по шаблону. $^R позволяет выполнить код и запомнить результат выполнения этого кода для использования далее в шаблоне или после сопоставления.

При обработке шаблона механизм регулярных выражений может встретить несколько выражений (?{ *CODE* }). Поэтому механизм запоминает каждое значение $^R и при необходимости осуществить возврат восстанавливает соответствующее прежнее значение $^R. Иными словами, $^R имеет в шаблоне динамическую область видимости, аналогично $1 и прочим переменным.

Поэтому $^R представляет не просто результат выполнения последнего фрагмента кода в шаблоне. Это результат выполнения последнего фрагмента кода на пути *к успешному сопоставлению*. Иначе говоря, в случае неудачи будет восстановлено прежнее значение $^R.

Если шаблон (?{ CODE }) действует непосредственно как условие подшаблона (?(*COND*) *IFTRUE*|*IFFALSE*), значение $^R не устанавливается.

$LAST_SUBMATCH_RESULT

$^N **[DYN,RO]** Текст соответствия последней закрытой группе (закрывающая скобка которой находится правее всех остальных), в последней успешной операции поиска по шаблону.

В основном используется внутри блоков (?{...}) для проверки только что совпавшего текста. Например, чтобы сохранить текст соответствия в переменной (в дополнение к переменным $1, $2 и другим), замените (...) на:

```
(?:(PATTERN)(?{ $var = $^N }))
```

Это избавит от необходимости выяснять, какой паре скобок соответствует искомое соответствие.

Данная переменная появилась в Perl версии v5.8.

Мнемоника: (возможно) вложенные[1] скобки, закрытые самыми последними.

$LIST_SEPARATOR

$" **[ALL] Когда массив или его срез интерполируются в строке в двойных кавычках (или аналогичной конструкции), эта переменная задает строку-разделитель для отдельных элементов. По умолчанию равна пробелу. (Мнемоника: надо думать, очевидная.)**

$^M **[ALL] По умолчанию ошибка нехватки памяти не перехватывается. Однако если при компиляции *perl* была зарезервирована возможность применения** $^M**, ей можно отвести роль аварийного пула памяти. Если Perl скомпилирован с *-DPERL_EMERGENCY_SBRK*, а также использует Perl-реализацию** malloc**, операция**

```
$^M = "a" x (1 << 16);
```

выделит буфер объемом в 64 килобайта для аварийного использования. О том, как включить этот режим, читайте в файле *INSTALL* в каталоге дистрибутива исходного кода Perl. Чтобы не поощрять случайное применение этой функции, для данной переменной не определено длинное имя в use English **(а какая мнемоника, мы вам не скажем).**

$MATCH

$& **[DYN,RO] Строка, найденная при последнем успешном поиске по шаблону в активной на данный момент динамической области видимости. (Мнемоника: как** & **в некоторых редакторах.)**

Использование этой переменной в программе отрицательно сказывается на производительности всех операций сопоставления с регулярными выражениями. Чтобы избежать этого, те же самые подстроки можно извлекать с помощью @-**. Начиная с версии v5.10 появилась возможность использовать флаг** /p **в регулярных выражениях, что дает нам переменную** ${^MATCH}**, играющую ту же роль для конкретной операции поиска по шаблону.**

${^MATCH}

[DYN,RO] Действует так же, как $& ($MATCH)**, но не снижает производительность. Гарантируется, что эта переменная будет содержать определенное значение, только если шаблон скомпилирован или выполнен с модификатором** /p.

Эта переменная появилась в Perl версии v5.10.

$OSNAME

$^O **[ALL] Содержит название платформы (обычно – операционной системы), для которой был скомпилирован текущий исполняемый модуль *perl*. Дешевая альтернатива извлечению данной информации из модуля** Config.

$OS_ERROR

$ERRNO

$! **[ALL] При использовании в числовом контексте возвращает текущее значение последней ошибки системного вызова – и, конечно, здесь действительны все обычные предостережения. (Это значит, что** $! **может и не содержать нужной вам информации, и на ее значение можно полагаться только в случае, если она**

---

[1] От английского слова Nested, обозначающего вложенность. – *Прим. лит. ред.*

содержит особое значение, однозначно указывающее на системную ошибку.) В строковом контексте $! возвращает соответствующее сообщение о системной ошибке. Можно присвоить $! свой номер ошибки, чтобы $! вернула строку, соответствующую этой ошибке, или чтобы установить значение выхода для die. См. также описание модуля Errno. (Мнемоника: что там грохнулось?)

%OS_ERROR

%ERRNO

%! [ALL] Каждый элемент %! имеет истинное значение, только если переменная $! установлена в значение этого элемента. Например, $!{ENOENT} имеет истинное значение, только когда текущим значением переменной $! является значение ENOENT, т.е. если последней была ошибка «No such file or directory» (нет такого файла или каталога) (или ее эквивалент: не все операционные системы и не все языки программирования дают именно такую ошибку). Проверить наличие того или иного ключа в конкретной системе можно с помощью выражения exists $!{*SOMEKEY*}; получить список всех допустимых ключей можно с помощью %!. Дополнительную информацию вы найдете в документации модуля Errno, а также в описании переменной $! выше.

Эта переменная появилась в Perl версии v5.005.

autoflush *HANDLE EXPR*

$AUTOFLUSH

$| [FHA] Будучи установленной в истинное значение, вызывает очистку буфера после каждой команды print, printf и write для выбранного в данный момент дескриптора файла вывода. (Мы называем это *буферизацией команд*. Вопреки распространенному мнению, установка этой переменной не отключает буферизацию.) Значением по умолчанию является «ложь», что для многих систем означает, что STDOUT будет буферизоваться построчно, если вывод происходит на терминал, и блочно – в ином случае, даже для каналов и сокетов. Установка этой переменной полезна, если вывод производится в канал, например, если запускается сценарий Perl под *rsh*(1), и требуется наблюдать за выводом в реальном времени. Если выходной буфер выбранного в данный момент дескриптора файла содержит еще не выведенные данные, в момент, когда эта переменная получит истинное значение, в качестве побочного эффекта этот буфер будет немедленно «вытолкнут». Примеры управления буферизацией для дескрипторов файлов, отличных от STDOUT, вы найдете в описании формы select с одним аргументом. (Мнемоника: когда нужно, чтобы каналы качали «от души».)

Эта переменная не влияет на буферизацию ввода, о которой сказано в описании функции getc в главе 27 или примере для модуля POSIX.

$OUTPUT_FIELD_SEPARATOR

$, [ALL] Разделитель полей вывода для print. Обычно print просто выводит указанный список элементов, не разделяя их. Устанавливайте эту переменную так же, как переменную *awk* OFS для задания символов, выводимых между полями. (Мнемоника: то, что выводится, когда в операторе print есть запятая (,).)

$OUTPUT_RECORD_SEPARATOR

$\ [ALL] Разделитель выходных записей (фактически завершающий символ) для print. Обычно print просто выводит перечисленные через запятую поля

без символа перевода строки или разделителя записей в конце. Присваивайте этой переменной последовательность символов для вывода в конце записи, при каждом вызове `print`, по аналогии с переменной *awk* `ORS`. (Мнемоника: `$\` устанавливается вместо `"\n"`, добавляемого в конце вывода. Кроме того, она похожа на /, но это то, что мы получаем «обратно» от Perl.) См. также описание ключа командной строки *-l* (от «line») в главе 17.

`%OVERLOAD`

[NOT,PKG] Элементы этого хеша устанавливаются директивой `use overload` с целью реализации перегрузки операторов для объектов класса текущего пакета. См. главу 13.

`$PERLDB`

`$^P` [NOT,ALL] Внутренняя переменная для включения отладчика Perl (*perl -d*).

`$PERL_VERSION`

`$^V` [ALL] Номер версии, подверсии и ревизии интерпретатора Perl. Эта переменная появилась в Perl версии v5.6.0 – в более ранних версиях будет иметь неопределенное значение. До версии v5.10.0 переменная `$^V` содержала v-строку.

`$^V` можно использовать для определения версии интерпретатора Perl, под управлением которого выполняется сценарий. Например:

```
warn "Hashes not randomized!\n" unless $^V && $^V gt v5.8;
```

Преобразовать значение `$^V` в строку можно с помощью спецификатора формата `"%vd"` функции `sprintf`:

```
printf "version is v%vd\n", $^V; # Версия Perl
```

В более новых версиях Perl это делается автоматически:

```
$ perl -E 'say $^V'
v5.14.0

$ perl -E 'say $^V > 5.10.1'
1
```

Читайте в описании `use` *`VERSION`* и `require` *`VERSION`* о том, как удобнее завершить работу, если используемый интерпретатор Perl старше, чем вы рассчитывали. В описании `$]` содержится информация о том, как изначально представлялась версия Perl.

Мнемоника: используйте `^V` для управления версиями.

`$POSTMATCH`

`$'` [DYN,RO] Строка, следующая за тем, что было найдено в последней удачной операции поиска по шаблону в активной на данный момент динамической области видимости. (Мнемоника: ' часто следует за цитируемой строкой.) Пример:

```
$_ = "abcdefghi";
/def/;
print "$`:$&:$'\n"; # выведет abc:def:ghi
```

Из-за динамической области видимости Perl не знает, каким шаблонам потребуется сохранять свои результаты в этих переменных, поэтому упоминание `$`` или `$'` где-либо в программе снижает производительность всех опера-

ций поиска по шаблону в программе. В небольших программах это не вызывает особых проблем, но при написании кода модуля, предназначенного для повторного использования, применения этой пары лучше избегать. Приведенный выше пример можно переделать, чтобы избежать урона общей производительности:

```
$_ = "abcdefghi";
/(.*?)(def)(.*)/s;  # /s на случай, если $1 содержит перевод строки
print "$1:$2:$3\n"; # выведет abc:def:ghi
```

${^POSTMATCH}

[DYN,RO] Подобна переменной $' ($POSTMATCH)**, но не снижает производительность. Гарантируется, что эта переменная будет содержать определенное значение, только если шаблон скомпилирован или выполнен с модификатором** /p.

Эта переменная появилась в Perl версии v5.10.

$PREMATCH

$` **[DYN,RO] Строка, предшествующая тому, что было найдено в последней удачной операции поиска по шаблону в активной на данный момент динамической области видимости. (Мнемоника: ` часто предшествует цитируемой строке.) См. выше замечание о производительности для** $'**.**

${^PREMATCH}

[DYN,RO] Подобна переменной $` ($PREMATCH)**, но не снижает производительность. Гарантируется, что эта переменная будет содержать определенное значение, только если шаблон скомпилирован или выполнен с модификатором** /p.

Эта переменная появилась в Perl версии v5.10.

$PROCESS_ID

$$ **[ALL] Номер процесса (PID) Perl, выполняющего данный сценарий. Эта переменная автоматически обновляется при вызове** fork**. На практике можно даже установить** $$ **самостоятельно; при этом, однако, PID процесса не изменится. Это ловкий фокус. (Мнемоника: та же, что в различных интерпретаторах команд.)**

Будьте осторожны с $$**; в некоторых случаях она может быть ложно истолкована как разыменование:** $$alphanum**. В такой ситуации пишите** ${$}alphanum**, чтобы отличить ее от** ${$alphanum}**.**

$PROGRAM_NAME

$0 **[ALL] Содержит имя файла выполняющегося сценария Perl. Присваивание переменной** $0 **представляет собой волшебство: оно пытается изменить область аргументов, о которой обычно сообщает программа *ps*(1). Это полезнее применять для индикации состояния, чем для сокрытия выполняемой программы. Но это работает не на всех системах. (Мнемоника: та же, что в *sh*, *ksh*, *bash* и т.д.)**

В многопоточных сценариях Perl координирует работу потоков выполнения так, что любой из них может изменять свою копию $0**, и эти изменения доступны команде *ps* (предполагается, что система позволяет это). Имейте в виду, что с точки зрения других потоков выполнения значение** $0 **не изменяется, поскольку эти потоки имеют собственные копии переменной.**

Если программа запущена с ключом *-e* **или** *-E*, $0 **будет содержать строку** "-e"**.**

$REAL_GROUP_ID

$(**[ALL] Реальный ID группы (GID) данного процесса. Если платформа поддерживает одновременное членство в нескольких группах, `$(` хранит разделенный пробелами список ваших групп. Первое число – это значение, которое возвращает *getgid*(2), а последующие представляют собой числа, возвращаемые *getgroups*(2), одно из которых может совпадать с первым числом.**

Однако для установки реального GID через присваивание `$(` следует использовать единственное число. Поэтому значение, которое дает `$(`, *не* должно присваиваться обратно `$(` без принудительного преобразования его в число, например, путем сложения с нулем. Это связано с тем, что реальная группа может быть только одна. См. описание `$)` (`$EFFECTIVE_GROUP_ID`), позволяющей устанавливать несколько текущих групп.

(Мнемоника: круглые скобки предназначены для *группировки* объектов. Реальный GID – это группа, которую мы *покинули* (*left*[1]), выполняя setgid.)

$REAL_USER_ID

$< **[ALL] Реальный ID пользователя (UID) данного процесса, как он возвращается системным вызовом *getuid*(2). Можно ли его изменить и как, зависит от особенностей реализации системы – см. примеры в `$>` (`$EFFECTIVE_USER_ID`). (Мнемоника: это UID, от которого мы пришли, выполняя setuid.)**

%SIG

[ALL] Хеш, используемый для установки обработчиков различных сигналов. (См. раздел «Сигналы» главы 15.) Например:

```
sub handler {
    my $sig = shift; # 1-й аргумент является именем сигнала
    syswrite STDERR, "Перехват SIG$sig -- завершение работы\n";
                     # Избегайте использования стандартного ввода/вывода
                     # в асинхронных обработчиках, чтобы избежать дампов памяти
                     # (Даже конкатенация строк опасна.)
    close LOG;       # Обращается к стандартному I/O, поэтому может
                     # завершиться аварийно!
    exit 1           # Но так как мы выходим, можно попробовать.
}

$SIG{INT}  = \&handler;
$SIG{QUIT} = \&handler;
...
$SIG{INT}  = "DEFAULT"; # восстановить действие по умолчанию
$SIG{QUIT} = "IGNORE";  # игнорировать SIGQUIT
```

Хеш `%SIG` содержит неопределенные значения, соответствующие сигналам, для которых не установлены обработчики. Обработчик можно задать как ссылку на подпрограмму или строку. Строковое значение, не являющееся одним из специальных действий `"DEFAULT"` или `"IGNORE"`, представляет собой имя функции, которая, если ее имя не квалифицировано пакетом, считается принадлежащей к пакету `main`. Вот еще несколько примеров:

---

[1] Имеется в виду левая скобка. В английском языке слово «левая» и глагол «покинуть» в форме прошедшего времени пишутся одинаково: left. – *Прим. ред.*

```
$SIG{PIPE} = "Plumber"; # годится, предполагает main::Plumber
$SIG{PIPE} = \&Plumber; # отлично, использовать Plumber из текущего пакета
```

С помощью хеша %SIG можно также установить некоторые внутренние обработчики. Подпрограмма, на которую указывает $SIG{__WARN__}, вызывается перед тем, как будет выведено предупреждение. Сообщение передается в первом аргументе. Существование обработчика __WARN__ вызывает подавление обычного вывода предупреждений на STDERR. Такой обработчик можно использовать для сохранения предупреждений в переменной или чтобы сделать предупреждения фатальными ошибками, например:

```
local $SIG{__WARN__} = sub { die $_[0] },
eval $proggie;
```

Это аналогично высказыванию

```
use warnings qw/FATAL all/;
eval $proggie;
```

за исключением того, что в первом случае область видимости динамическая, а во втором – лексическая.

Подпрограмма, на которую указывает $SIG{__DIE__}, превращает исключение-лягушку в исключение-принцессу с помощью волшебного поцелуя, что часто не работает. Наилучшее применение этого обработчика – некая завершающая обработка перед выходом для умирающей программы, которая должна скончаться от необработанного исключения. От смерти это не спасет, но еще немного подышать получится.

Сообщение исключительной ситуации передается в первом аргументе. При возврате из обработчика __DIE__ дальнейшая обработка исключения происходит так, как если бы обработчика не было, если только подпрограмма-обработчик не завершится через goto, выход из цикла или die. Во время вызова обработчик __DIE__ явным образом отключается, чтобы была возможность вызвать настоящий die из обработчика __DIE__. (Если его не отключить, то обработчик будет рекурсивно вызывать себя до бесконечности.) Обработчик для $SIG{__WARN__} действует аналогично.

Устанавливать $SIG{__DIE__} нужно только в основной программе, но не в модулях. Это связано с тем, что в настоящее время даже перехватываемые исключения все же запускают обработчик $SIG{__DIE__}. Делать это настоятельно не рекомендуется, поскольку обработчик может нарушить работу невинных модулей, не ожидающих, что предполагаемое ими поведение в исключительных ситуациях таинственным образом изменится. Используйте эту возможность только в качестве последнего средства, и, если приходится это делать, всегда ставьте впереди local, чтобы сократить опасный период.

Если вы привыкли к языкам программирования, которые реагируют на необработанные исключения заполнением экрана стеком вызовов, то можете заставить Perl делать то же самое, поместив следующие строки в модуль main своей программы:

```
use Carp;
$SIG{__DIE__} = sub { confess "$0: UNCAUGHT EXCEPTION: @_" unless $^S };
```

Не пытайтесь построить на этой возможности механизм обработки исключений. Вместо этого для перехвата исключений используйте eval {}. Например, чем использовать обработчик __DIE__, оформите основную программу в виде подпрограммы и оберните ее стандартным обработчиком исключений – обычным eval *BLOCK*:

```
use Carp;
eval {
    function_that_does_everything();
    1;
} || confess "$0  Caught unexpected exception: $@";
```

STDERR

[ALL] Специальный дескриптор файла для стандартного устройства вывода ошибок во всех пакетах.

STDIN

[ALL] Специальный дескриптор файла для стандартного устройства ввода во всех пакетах.

STDOUT

[ALL] Специальный дескриптор файла для стандартного устройства вывода во всех пакетах.

$SUBSCRIPT_SEPARATOR

$; [ALL] Разделитель индексов для эмуляции многомерных хешей. Если вы ссылаетесь на элемент хеша как

```
$foo{$a,$b,$c}
```

то в действительности это значит:

```
$foo{join($;, $a, $b, $c)}
```

Но не пишите:

```
@foo{$a,$b,$c} # срез, обратите внимание на @
```

что означает:

```
($foo{$a},$foo{$b},$foo{$c})
```

Значением по умолчанию является "\034" – то же, что SUBSEP в *awk*. Обратите внимание, что, если в ключах содержатся двоичные данные, то безопасного значения для $; может не найтись. (Мнемоника: запятая – разделитель синтаксических индексов – это половина точки с запятой. Коряво, конечно, но $, уже занята для более важных задач.)

Хотя мы не объявили эту функцию устаревшей, сейчас лучше использовать «настоящие» многомерные хеши, т.е. $foo{$a}{$b}{$c}, вместо $foo{$a,$b,$c}. Однако может оказаться, что поддельные хеши легче сортируются и проще в использовании в качестве DBM-файлов.

$SYSTEM_FD_MAX

$^F [ALL] Максимальный «системный» номер файла, обычно 2. Системные номера файлов передаются новым программам во время exec, а номера с бо́льшими значениями – нет. Кроме того, во время выполнения open системные номера

файлов сохраняются, даже если вызов этой функции завершается неудачей. (Обычные указатели файлов закрываются перед попыткой выполнения open и остаются закрытыми при ее отказе.) Обратите внимание, что статус дескриптора файла в отношении закрытия при exec определяется согласно значению $^F во время выполнения open, а не во время выполнения exec. Этого можно избежать, сначала временно присвоив $^F заведомо большое значение:

```
{
    local $^F = 10_000;
    pipe(HITHER,THITHER) or die "can't pipe: $!";
}
```

${^TAINT}

[ALL,RO] Эта доступная только для чтения переменная отражает состояние режима проверки меченых данных: включено, выключено или простой вывод предупреждений:

0	Режим проверки меченых данных выключен (по умолчанию).
1	Режим проверки меченых данных включен, обычно из-за того, что программа была запущена с ключом *-T*.
-1	Только выводить предупреждения, включается ключами командной строки *-t* и *-TU*.

Эта переменная появилась в Perl версии v5.8.

${^UNICODE}

[XXX,ALL] Эта переменная отражает некоторые внутренние настройки Юникода в Perl. Ей можно присвоить числовое значение на этапе запуска Perl ключом командной строки *-C* или посредством переменной среды PERL_UNICODE; после этого она будет доступна только для чтения.

Эта переменная появилась в Perl версии v5.8.2.

${^UTF8CACHE}

[NOT, ALL] Внутренняя переменная, управляющая состоянием внутреннего механизма кэширования смещения в UTF-8:

1	Включен (по умолчанию).
0	Выключен.
-1	Для отладки механизма кэширования посредством сравнения всех результатов с линейным сканированием и вывода сообщений обо всех несоответствиях. Устанавливается ключом командной строки *-Ca*.

Эта переменная появилась в Perl версии v5.8.9.

${^UTF8LOCALE}

[NOT,ALL] Указывает, была ли определена настройка кодировки UTF-8 в национальных настройках системы при запуске Perl. Эта информация используется Perl для перекодировки UTF-8 в локальную кодировку, устанавливается ключом командной строки *-CL*.

Эта переменная появилась в Perl версии v5.8.8.

$VERSION

[PKG] Обращение к этой переменной происходит, когда задается минимальная допустимая версия модуля, как в `use SomeMod 2.5`. Если `$SomeMod::VERSION` меньше этого значения, возникает исключение. Эту переменную просматривает метод `UNIVERSAL->VERSION`, поэтому можно определить собственную функцию `VERSION` в текущем пакете, если нужно что-либо иное, кроме режима по умолчанию. См. главу 12.

$WARNING

$^W [ALL] Текущее логическое значение ключа глобального предупреждения (не путать с глобальным потеплением, относительно которого слышно много глобальных предупреждений).[1] См. также описание директивы `warnings` в главе 29 и ключей командной строки *-W* и *-X* для предупреждений с лексической областью видимости, на которые данная переменная не влияет. (Мнемоника: значение связано с ключом *-w*.)

${^WARNING_BITS}

[NOT,ALL] Текущий набор проверок, активированных директивой `use warnings`. Подробности см. в описании `warnings` в главе 29.

${^WIDE_SYSTEM_CALLS}

[ALL] Глобальный флаг, разрешающий всем системным вызовам, осуществляемым Perl, использовать системный API обработки широких символов (wide characters), если таковой имеется. Включить эту возможность можно из командной строки посредством ключа *-C*. Начальное значение обычно `0` для совместимости с Perl версий до 5.6, но Perl может автоматически назначить этой переменной значение `1`, если система предоставляет пользовательское значение по умолчанию (например, через `$ENV{LC_CTYPE}`). Прагма `bytes` всегда переопределяет действие этого флага в текущей лексической области видимости.

${^WIN32_SLOPPY_STAT}

[ALL] Если эта переменная имеет истинное значение, функция `stat` не будет пытаться открыть файл при выполнении сценария в Windows. Это означает невозможность определить счетчик ссылок, и атрибуты файла могут отличаться от действительных при наличии дополнительных жестких ссылок на файл. С другой стороны, отказ от открытия файла существенно увеличивает скорость выполнения, особенно для файлов на сетевых устройствах.

Эта переменная может быть установлена в файле *sitecustomize.pl* с локальными настройками Perl, чтобы обеспечить такой «неполноценный» режим работы функции `stat` по умолчанию. См. описание ключа *-f* в разделе «Command Switches» страницы *perlrun*, где приводятся дополнительные сведения о локальных настройках.

Эта переменная появилась в Perl версии v5.10.

---

[1] В оригинале обыгрывается созвучие выражений «global warning» и «global warming». – *Прим. ред.*

26

Форматы

Perl славится своими богатыми возможностями манипулирования текстом – именно возможности извлечения (Extraction) сделали его таким популярным.[1] Perl также существенно упрощает форматирование строк при создании отчетов (Report). В этой главе рассматриваются функции printf и sprintf, pack и unpack, а также форматы, исторические предназначавшиеся для печати форматированных отчетов на принтере, но не потерявшие своей актуальности и в новом тысячелетии.

Форматы строк

Perl может производить форматированные строки, следуя обычным соглашениям для функций *printf* и *sprintf* из библиотеки языка C. Версия функции sprintf в Perl возвращает строку, а версия printf выводит ее в дескриптор файла, указанный явно или используемый по умолчанию:

```
sprintf FORMAT, LIST
printf FORMAT, LIST
printf FILEHANDLE FORMAT, LIST
```

Функция sprintf обрабатывает свои аргументы немного иначе. Первый аргумент она всегда интерпретирует как скаляр, даже если это массив. Вероятно, вам нужно что-то другое, поскольку в последнем случае функция интерпретирует @array в скалярном контексте и просто выведет число элементов в массиве:

```
my @array = ( '%d %d %d', 1, 2, 3 );
sprintf @array;
```

Функция printf обрабатывает аргументы иначе потому, что ее первым необязательным аргументом служит дескриптор *FILEHANDLE*.

---

1 Напомню, что одним из толкований акронима Perl является: «Practical Extraction and Report Language». – *Прим. перев.*

Строка *FORMAT* – это текст, содержащий внедренные спецификаторы полей, вместо которых подставляются элементы списка *LIST*. Это – одна из особенностей, заимствованных языком Perl из C, поэтому за описанием основных принципов можно смело обращаться к страницам *sprintf*(3) и *printf*(3) справочного руководства системы.

В Perl используется собственная реализация форматирования в `sprintf` – она имитирует функцию `sprintf` в C, но не использует ее.[1] Поэтому нестандартные расширения вашей версии *sprintf*(3) недоступны в Perl.

Функция `sprintf` в Perl поддерживает все распространенные спецификаторы форматов, перечисленные в табл. 26.1.

Таблица 26.1. Спецификаторы форматов для sprintf

Спецификатор	Значение
%%	Литерал знака процента
%b	Целое без знака в двоичной системе счисления
%B	Аналогично %b, но использует «B» в верхнем регистре с флагом #
%c	Символ с указанным кодом
%d	Целое со знаком, в десятичной системе счисления
%e	Число с десятичной запятой, в экспоненциальном представлении с префиксом экспоненты «e» в нижнем регистре
%E	Аналогично %e, но с префиксом экспоненты «E» в верхнем регистре
%f	Число с десятичной запятой, в фиксированном десятичном формате
%g	Число с десятичной запятой, в формате %e или %f
%G	Аналогично %g, но с префиксом экспоненты «E» в верхнем регистре, если применимо
%h	Короткое целое или короткое целое без знака языка C, в зависимости от компилятора, с помощью которого был собран *perl*
%n	Сохраняет число выведенных до сих пор символов char в следующую переменную в списке аргументов
%o	Целое без знака, в восьмеричной системе счисления
%p	Указатель (выводит адрес значения в шестнадцатеричной системе счисления)
%s	Строка неопределенной длины
%u	Целое без знака, в десятичной системе счисления
%x	Целое без знака, в шестнадцатеричной системе счисления, символами нижнего регистра
%X	Целое без знака, в шестнадцатеричной системе счисления, символами верхнего регистра

[1] За исключением вещественных чисел, но даже в этом случае допустимы только стандартные модификаторы.

В табл. 26.2 перечислены те же спецификаторы форматов, но сгруппированные по типам выводимых значений.

Таблица 26.2. Спецификаторы форматов по типам выводимых значений

Тип	Спецификаторы
Целые числа	%b %B %d %h %o %p %u
Вещественные числа	%e %E %f %g %G
Строки	%c %s

Для некоторых спецификаторов числовых форматов можно указать, как sprintf должна интерпретировать число, не полагаясь на размеры, предложенные компилятором. См. табл. 26.3.

Таблица 26.3. Спецификаторы числовых преобразований в sprintf

Спецификатор	Значение
hh	char или unsigned char в языке C (v5.14 и выше)
h	short или unsigned short в языке C (v5.14 и выше)
j	тип intmax_t в языке C (v5.14 и выше, для компиляторов C99)
l	long или unsigned long в языке C
q, L, ll	long long, unsigned long long или quad в языке C (компилятор должен поддерживать тип quad)
t	ptrdiff_t в языке C (v5.14 и выше)
v	Интерпретирует строку как вектор целых чисел, выводит их как целые, разделяя точками или произвольной строкой, полученной из списка аргументов, если перед флагом стоит *
z	size_t в языке C (v5.14 и выше)

Для обратной (мы подразумеваем именно «направленной назад») совместимости Perl допускает факультативные, но широко распространенные преобразования, перечисленные в табл. 26.4. Мы отделили их от спецификаторов, перечисленных в табл. 26.1, в надежде, что вы не будете пользоваться ими.

Таблица 26.4. Синонимы числовых преобразований для обратной совместимости

Спецификатор	Значение
%i	Синоним %d
%D	Синоним %ld
%U	Синоним %lu
%O	Синоним %lo
%F	Синоним %f

Между % и символом преобразования допускается указывать дополнительные атрибуты, перечисленные в табл. 26.5 и управляющие интерпретацией формата:

Таблица 26.5. Модификаторы формата для sprintf

Модификатор	Значение
пробел	Пробел перед положительным числом
+	Плюс перед положительным числом
-	Выравнивание по левому краю поля
0	Использовать нули, а не пробелы, для выравнивания по правому краю поля
#	Использовать в качестве префикса ненулевых восьмеричных чисел "0", ненулевых шестнадцатеричных "0x", ненулевых двоичных чисел "0b"
*	Использовать значение следующего аргумента в качестве ширины поля
число$	Использовать значение аргумента в позиции *число*
**число*$	Использовать значение аргумента в позиции *число* в качестве ширины поля
число	Минимальная ширина поля (не существует эквивалента для обозначения максимальной ширины поля)
.*число*	«Точность»: количество цифр после десятичной запятой для вещественных чисел, максимальная длина для строки, минимальная длина для целого

Ниже мы приведем несколько примеров. Добавление пробела перед буквой спецификатора обеспечивает вывод ровно одного пробела перед числом, независимо от его размера:

```
printf "<% d>", 1, # "< 1>"
```

Добавление символа + обеспечивает вывод знака положительного числа, даже если это число 0:[1]

```
printf "<%+d>", 1; # "<+1>"
printf "<%+d>", 2, # "<+2>"
```

Добавление пробела и знака + в паре в любом порядке обеспечивает вывод знака + перед положительным числом:

```
printf "<%+ d>", 3; # "<+3>"
printf "<% +d>", 5; # "<+5>"
```

Определение точности для целого числа обеспечивает дополнение нулями слева. Знак + не учитывается при расчете ширины поля:

```
printf "<%.5d>",   8; # "<00008>"
printf "<%+.5d>", 13; # "<+00013>"
```

Если ширина поля оказывается больше, дополнение нулями будет выполнено только до ширины, указанной в атрибуте точности. Значения можно выравнивать по левому или по правому краю:

```
printf "<%-10.6d>", 21; # "<000021    >"
printf "<%10.6d>",  34; # "<    000034>"
printf "<%010.6d>", 55; # "<    000055>"
printf "<%+10.6d>", 89; # "<   +000089>"
```

[1] Это отличается от понятий приближения предела к нулю с разных сторон, 0^+ и 0^-.

Все это в равной степени относится ко всем целочисленным форматам.

Строки по умолчанию выравниваются по правому краю, но, применив знак «минус», можно добиться выравнивания строк по левому краю:

```
printf "<%6s>",  144; # "<   144>"
printf "<%-6s>", 233; # "<233   >"
```

Начальный 0 обеспечивает заполнение пустых позиций нулями, но только слева, даже если значение не является числом:

```
printf "<%06s>",  377;   # "<000377>"
printf "<%-06s>", 610;   # "<610   >" - ведущие нули не выводятся
printf "<%06s>", "Perl"; # "<00Perl>"
```

Значение ширины удобно использовать для выравнивания строк по колонкам фиксированной ширины. Однако printf интерпретирует значение ширины только как минимальное значение. Она не производит усечение строк:

```
printf "<%5s>", 'Amelia'; # "<Amelia>", выводятся все шесть символов
```

Если требуется обеспечить усечение строк, можно указать значение точности *число* после значения ширины поля:

```
printf "<%5.5s>", 'Camelia'; # "<Camel>", только пять символов
```

Ширина поля и количество допустимых символов могут не совпадать. Если в параметре точности будет указано больше символов, чем в параметре ширины поля, ширина поля будет игнорироваться:

```
printf "<%3.5s>\n", "Camelia"; # "<Camel>"
```

Обычно функция printf замещает спецификатор следующим неиспользованным аргументом, но с помощью параметра *число*$ можно явно указать, какой аргумент использовать. При использовании параметра *число*$ следует предпринять все меры, чтобы по ошибке не использовать интерполируемую строку или не экранировать знак доллара; в противном случае, Perl будет думать, что вы пытаетесь интерполировать переменную в этом месте.

```
printf '%2$d %1$d', 12, 34;      # "34 12"
printf '%3$d %d %1$d', 1, 2, 3; # "3 1 1"
```

Этот прием позволяет повторно использовать аргументы:

```
printf '%2$d %1$d %2$d', 12, 34; # "34 12 34"
```

Иногда ширина поля неизвестна заранее, поэтому предоставляется возможность передать ее в аргументе с помощью *. Этот флаг извлекает значение ширины из аргумента, предшествующего аргументу со строкой:

```
printf "<%*s>", 6, "Perl"; # "<  Perl>"
```

Если аргумент имеет отрицательное значение, выполняется выравнивание по левому краю:

```
printf "<%*s>", -6, "Perl"; # "<Perl  >"
```

Для определения номера аргумента с шириной поля можно использовать параметр вида *число*$:

```
printf '<%*2$s>', "a", 6; # "<     a>"
```

Если потребуется передать в аргументах ширину поля и максимальное количество выводимых символов, используйте * в обеих позициях:

```
printf "<%*.*s>\n", 10, 5, "Camelia"; # "<     Camel>"
```

но номер аргумента можно указать только для значения ширины:

```
printf '<%*2$.*s> , "Camelia", 10, 5, # "<   Camelia>"
printf '<%*.*2$s>', "Camelia", 10, 5; # "<%*.*2$s>"
```

Флаг # обеспечивает вывод перед числом дополнительных символов, обозначающих систему счисления, но только когда значение не равно 0 (в этом случае система счисления не имеет значения):

```
printf "<%#o>",  37; # "<045>"

printf "<%#x>",  42. # "<0x2a>"
printf "<%#X>",  42; # "<0X2A>'

printf "<%#b>", 137, # "<0b10001001>"
printf "<%#B>", 137; # "<0B10001001>"
```

Когда в спецификаторе %o указаны и флаг #, и значение точности, точность не учитывает ведущий «0»:

```
printf "<%#.5o>",   0377; # "<00377>"
printf "<%#.5o>", 010755; # "<010755>"
printf "<%#.0o>",      0; # "<0>"
```

Для вещественных значений (%e, %f и %g) можно указать количество десятичных знаков после запятой с помощью параметра *.число*. **Если используется параметр ширины поля, точность указывается вслед за ним. Имейте в виду, что этот параметр приводит к округлению чисел перед выводом:**

```
printf "<%f>",   3.14159265; # "<3.141593>"
printf "<%.1f>", 3.14159265; # "<3.1>"
printf "<%.0f>", 3.14159265; # "<3>"

printf "<%e>",   6.62606857e-34; # "<6.626069e-34>"
printf "<%.1e>"  1.05457148e-34: # "<1.1e-34>'
```

Если задействована прагма use locale (см. главу 29) и была вызвана функция POSIX::setlocale, в качестве разделителя целой и дробной части будет использоваться символ, определяемый региональными настройками:

```
use POSIX;
use locale;

POSIX::setlocale(LC_NUMERIC, "fr_FR");
printf "<%f>", 3.1415926; # "<3,141593>"
```

Спецификатор %g использует системные настройки, поэтому в разных системах могут получаться немного разные результаты:

```
printf "<%g>",    1 << 31; # "<2.14748e+09>"
printf "<%.5g>",  1 << 31; # "<2.1475e+09>"
printf "<%.10g>", 1 << 31; # "<2147483648>"
```

Выбор количества мест для отображения экспоненты со значением меньше 100 зависит от системы. В некоторых системах значение может дополняться нулем слева:

```
printf "<%g>", 1 << 31; # "<2.14748e+009>", может быть
```

Спецификатор v отличается от других. Он считает свой строковый аргумент вектором, где каждый элемент является целым числом, которое требуется вывести с указанным форматированием. Он выводит целые числа через точку. Использование спецификатора шестнадцатеричной системы счисления может пригодиться для вывода последовательностей кодов символов в стиле Юникода:

```
printf "<%vd>", "\x5\xE\x2"; # "5.14.2"

use utf8;
printf "<%vd>", "À%{"; # "<65.758.37.123>"
printf "<%vX>", "À%{"; # "<41.300.25.7B>"
printf "U+v%04x", "À%{"; # "U+0041.0300.0025.007B"
```

(Обратите внимание, что графема «À» выше состоит из двух кодов символов.)

Если нежелательно использовать точку между числами, можно передать свой разделитель в аргументе:

```
printf "<%*vX>", ":", "À%{";   # "<41:300:325:7B>"
printf "<%*2$vX>", "À%{", ":"; # "<41:300:25:7B>"
```

Графемы, в особенности состоящие из нескольких кодов, создают определенные проблемы. Как будет показано далее в этой главе, в ходе рассказа о двух других форматах, Perl дает неправильный ответ при вычислении ширины для данных в Юникоде, содержащих непечатаемые символы, комбинационные знаки и широкие символы. То же относится к параметру ширины поля в строках формата для printf и sprintf. В разделе «Графемы и нормализация» главы 6 приводится пример, демонстрирующий, как можно воспользоваться методом columns из модуля Unicode::GCString, чтобы обхитрить printf и заставить ее работать правильно, несмотря на описываемые недостатки. Суть стратегии состоит в том, чтобы предварительно определить правильную ширину поля с помощью умного метода columns, не рассчитывая, что бесхитростная функция printf осилит сложные вычисления.

Двоичные форматы

Если вы знакомы с традиционными языками программирования, то, вероятно, уже сталкивались с понятием записей или структур. В противоположность функции sprintf, основным назначением которой является создание строк, понятных человеку, функции pack и unpack предназначены для низкоуровневых преобразований и форматирования структур или записей данных простых типов в строковое представление (и обратно). Обе функции используют общий язык шаблонов с небольшими отличиями, описываемыми в следующем разделе.

pack

pack *TEMPLATE*, *LIST*

Эта функция принимает список *LIST* обычных значений Perl, преобразует их в строку байтов согласно шаблону *TEMPLATE* и возвращает эту строку. Список аргументов

при необходимости дополняется или обрезается, т.е. если задать меньше аргументов, чем требует *TEMPLATE*, pack будет интерпретировать недостающие аргументы как пустые значения, а если задать больше аргументов, чем требует *TEMPLATE*, лишние аргументы игнорируются. Нераспознанные элементы формата в *TEMPLATE* вызывают исключение.

Шаблон описывает структуру строки как последовательность полей. Каждое поле представлено одним символом, описывающим тип кодируемого в нем значения. Например, символ формата N задает четырехбайтовое целое число без знака с прямым (big-endian) порядком следования байтов.

Поля упаковываются в порядке, задаваемом шаблоном. Например, чтобы упаковать в строку однобайтовое целое без знака и значение с десятичной запятой с одинарной точностью, следует сказать:

```
$string = pack("Cf", 244, 3.14);
```

Первый байт возвращаемой строки имеет значение 244. В остальных байтах будет закодировано число 3.14 как число одинарной точности с десятичной запятой. Конкретная кодировка числа с десятичной запятой зависит от аппаратной архитектуры компьютера.

При упаковке нужно учитывать следующее:

- тип данных (например, целое число, число с десятичной запятой, строка);
- диапазон значений (например, помещаются ли ваши целые числа в один, два, четыре или, может быть, даже восемь байтов; упаковываете ли вы восьмиразрядные символы или символы Юникода);
- являются ли целые числа знаковыми или беззнаковыми;
- какая кодировка используется (родная, с обратным – little-endian – или с прямым – big-endian – порядком следования битов и байтов).

В табл. 26.6 перечислены символы формата и их значения. (В форматах встречаются и другие символы, которые будут описаны далее.)

Таблица 26.6. Символы шаблона для pack/unpack

Символ	Значение
a	Строка байтов, дополняемая нулевыми байтами слева
A	Строка байтов, дополняемая пробелами слева
b	Битовая строка в порядке возрастания старшинства разрядов в каждом байте (как vec)
B	Битовая строка в порядке убывания старшинства разрядов в каждом байте
c	Восьмиразрядное целое со знаком
C	Восьмиразрядное целое без знака; смотрите U для Юникода
d	Число с десятичной запятой двойной точности в родном для архитектуры формате
D	Число с десятичной запятой или число с десятичной запятой двойной длины в родном для архитектуры формате; числа с десятичной запятой двойной длины доступны, только если система поддерживает их и *perl* скомпилирован с их поддержкой

Символ	Значение
f	Число с десятичной запятой одинарной точности в родном для архитектуры формате
F	Число с десятичной запятой во внутреннем представлении Perl (NV) в родном для архитектуры формате
h	Шестнадцатеричная строка, младший полубайт впереди
H	Шестнадцатеричная строка, старший полубайт впереди
i	Целое со знаком в родном для архитектуры формате; не менее 32 разрядов, но может быть и больше, в зависимости от использованного компилятора С
I	Целое без знака в родном для архитектуры формате; не менее 32 разрядов, но может быть и больше, в зависимости от использованного компилятора С
j	Целое со знаком во внутреннем представлении Perl (IV)
J	Целое без знака во внутреннем представлении Perl (UV)
l	Длинное целое со знаком, всегда 32 разряда
L	Длинное целое без знака, всегда 32 разряда
n	16-разрядное короткое целое с «сетевым» (big-endian) порядком следования битов
N	32-разрядное длинное целое с «сетевым» (big-endian) порядком следования битов
p	Указатель на строку, завершающуюся пустым символом (null)
P	Указатель на строку фиксированной длины
q	64-разрядное целое (quad) со знаком
Q	64-разрядное целое (quad) без знака (только если система поддерживает 64-разрядные числа, и *perl* скомпилирован с их поддержкой)
s	Короткое целое (short) со знаком, всегда 16 разрядов
S	Короткое целое (short) без знака, всегда 16 разрядов
u	Строка в кодировке uuencode
U	Номер символа Юникода; в текстовом режиме выполняется преобразование в символ, а в двоичном – в соответствующий код UTF-8
v	16-разрядное целое с порядком следования битов «VAX» (little-endian)
V	32-разрядное целое с порядком следования битов «VAX» (little-endian)
w	Сжатое целое в формате BER (Binary Encoded Representation – представление в двоичном формате)
W	Значение символа без знака
x	Нулевой байт (проскочить на байт вперед)
X	Вернуться на байт
Z	Строка байтов, завершающаяся нулевым байтом (и дополненная нулевыми байтами)
@	Заполнить нулями до абсолютной позиции
.	Заполнить нулевыми байтами или обрезать до абсолютной позиции
(	Начало группы
)	Конец группы

Таблица 26.7. Модификаторы шаблонов для pack/unpack

Модификатор	Применяется к	Действие
!	iIlLsS	Принудительно устанавливает родные для архитектуры размеры
!	xX	Символы x и X действуют как символы выравнивания
!	nNvV	Целые без знака интерпретируются как целые со знаком
!	@.	Определяет позицию смещения во внутреннем представлении строки – опасно!
>	dDfFiIjJlLpPqQsS	Принудительно устанавливает прямой (big-endian) порядок следования байтов; может применяться к группам и подгруппам
<	dDfFiIjJlLpPqQsS	Принудительно устанавливает обратный (little-endian) порядок следования байтов; может применяться к группам и подгруппам

В шаблонах можно свободно располагать пробельные символы и комментарии. Комментарии начинаются символом # и простираются до первого символа перевода строки (если он есть) в шаблоне.

Повторение

За каждой буквой в шаблоне может следовать число, означающее *счетчик* (*count*), интерпретируемое как число повторений или некоторого рода длина, в зависимости от формата. Начнем с повторений, которые упаковывают символы или числа: c, C, d, D, f, F, i, I, j, J, l, L, n, N, p, q, Q, s, S, U, v, V, w и W.

Число после этих форматов представляет повторение, т.е. поле будет повторяться в строке, извлекая указанное число аргументов:

```
$out = pack 'C4', 192, 168, 1, 1  # \xC0\xA8\01\01
```

Необязательное число повторений заключается в квадратные скобки:

```
$out = pack 'C[4]', 192, 168, 1, 1  # \xC0\xA8\01\01
```

Буква в квадратных скобках обозначает длину соответствующего формата:

```
$out = pack 'C[N]', 192, 168, 1, 1  # \xC0\xA8\01\01
$out = pack 'C[s]', 192, 168, 1, 1  # \xC0\xA8
```

При нехватке аргументов лишние поля заполняются нулями:

```
$out = pack 'C4', 192, 168; # \xC0\xA8\00\00
```

Символ * соответствует всем оставшимся аргументам:

```
$out = pack 'C*', 192, 168, 1, 1, # \xC0\xA8\01\01
```

Остальные буквы меняют свой смысл с повторениями. Число после a, A или Z определяет длину, до которой должно быть дополнено поле:

```
$out = pack 'A10', 192, 168, 1, 1; # \xC0\xA8\01\01
```

Если число меньше длины строки, a и A обрезают ее:

```
$out = pack 'A4', 'Amelia'; # "Amel"
$out = pack 'a4', 'Amelia'; # "Amel"
```

В сочетании с * спецификаторы a и A воспроизводят поле с шириной, соответствующей аргументу:

```
$out = pack 'A*', 'Amelia'; # "Amelia"
$out = pack 'a*', 'Amelia'; # "Amelia"
```

Однако Z резервирует последнюю позицию для завершающего нулевого байта:

```
$out = pack 'Z4', 'Amelia'; # "Ame\000"
```

при условии, что число не равно нулю; в противном случае нулевой байт не добавляется:

```
$out = pack 'Z0', 'Amelia'; # ""
```

Комбинация Z* извлекает строку целиком, независимо от ее длины, и завершает ее нулевым байтом:

```
$out = pack 'Z*', 'Amelia'; # "Amelia\000"
```

Форматы b и B выводят указанное количество разрядов. b и B используют только один разряд (самый младший) из символов на входе и устанавливают только один разряд на выходе:

```
$out = pack 'B8', '10011101'; # 0b10011101
$out = pack 'b8', '10011101'; # 0b10111001
```

Форматы h and H действуют аналогично, используя счетчик в качестве числа полубайтов. Однако их особенность состоит в том, что символы, выглядящие как шестнадцатеричные цифры, интерпретируются как числа:

```
$out = pack 'h1', 'a';         # 0x0a
$out = pack 'H1', 'a';         # 0xa0
$out = pack 'H8', 'deadbeef'   # 0xdeadbeef
```

В противном случае используется младший полубайт:

```
$out = pack 'h2', '1',   # 0x01
$out = pack 'h2', 'one'; # 0x08
$out = pack 'H2', 'one'; # 0x80
```

Символ * с форматами h и H обеспечивает дополнение строки нулями до четного количества полубайтов:

```
$out = pack 'H*', 'deadbee'; # 0xdeadbee0
```

С форматом P счетчик определяет размер структуры для упаковки.

С форматом u счетчик обозначает длину строки в кодировке uuencode. Счетчик меньше 3 (или если используется *) интерпретируется как 45. Следующий формат:

```
$out = pack "u30", $some_string;
```

примет всю строку:

```
93VYE(')I;F<@=&\@<G5L92!T:&5M(&%L;
```

Но та же строка с меньшим значением счетчика:

```
$out = pack "u15", $some_string;
```

будет перенесена:

```
/3VYE(`)I;F<@=&\@<G5L
*92!T:&5M(&%L;```
```

Формат x не использует аргументы, но вставляет указанное число нулевых байтов. Модификатор * действует как значение 0:

```
$out = pack "H2 x h2",  "dead", "beef"; # 0xde00eb
$out = pack "H2 x3 h2", "dead", "beef"; # 0xde000000eb
$out = pack "H2 x* h2", "dead", "beef"; # 0xdeeb
```

Формат X не использует аргументы. Он возвращается на указанное число байтов назад, при условии, что не произойдет выход за начало строки. Символ * интерпретируется как 0:

```
$out = pack "H2 X h2",  "dead", "beef"; # 0xeb
$out = pack "H2 X3 h2", "dead", "beef"; # 0xde000000eb
$out = pack "H2 X* h2", "dead", "beef"; # 0xdeeb
```

Формат @ обрезает или дополняет до позиции относительно начала самой внутренней группы (или всей строки, если группы отсутствуют). Если упакованная строка получается длиннее счетчика, она обрезается до длины, указанной в счетчике. Если упакованная строка оказывается короче счетчика, она дополняется до длины, указанной в счетчике. В любом случае оставшаяся часть шаблона будет выполнять заполнение с новой позиции:

```
$out = pack 'A*',   'Amelia';            # Amelia
$out = pack 'A*@3', 'Amelia';            # Ame
$out = pack 'A*@3A*', 'Amelia', 'Camel'; # AmeCamel

$out = pack 'c@5', 137;                  # 0x8900000000
```

Символ * интерпретируется как 0, поэтому он обрезает все, следующее далее:

```
$out = pack 'A*@*A*', 'Amelia', 'Camel'; # Camel
```

Внутри группы обрезание или дополнение применяется только к группе:

```
$out = pack 'A(A@4A)A', 'A', 'B', 'C', 'D'; # "AB\000\000\000CD
$out = pack 'A(A@*A)A', 'A', 'B', 'C', 'D'; # "ACD
```

Формат . (точка) также обрезает или дополняет, но принимает номер позиции из списка аргументов. Счетчик повторений определяет начальную позицию применения: 0 – начать с текущей позиции, другое число определяет номер группы, а * соответствует началу строки:

```
# обрезать от начала строки
$out = pack 'A(A.*A)A', 'A', 'B', 1, 'C', 'D'; # 'ACD'

# дополнить от начала строки
$out = pack 'A(A.*A)A', 'A', 'B', 5, 'C', 'D'; # "AB\000\000\000CD"

# обрезать от начала строки
$out = pack 'A(A.1A)A', 'A', 'B', 1, 'C', 'D'; # "ABCD"

# дополнить от начала строки
```

```
$out = pack 'A(A.1A)A', 'A', 'B', 3, 'C', 'D'; # "AB\000\000\000\000CD"

# дополнить от текущей позиции
$out = pack 'A(A.0A)A', 'A', 'B', 0, 'C', 'D'; # "ABCD"
$out = pack 'A(A.0A)A', 'A', 'B', 2, 'C', 'D'; # "AB\000\000CD"
```

Другие модификаторы

Символ / позволяет упаковывать и распаковывать строки, в которых упакованная структура содержит счетчик байтов, за которым следует сама строка. Программист пишет: *length-item/string-item*. Элемент *length-item* может быть любой буквой шаблона для pack и описывает, как упаковано значение длины. Чаще всего используются форматы упаковки целых чисел, такие как n (для строк Java), w (для ASN.1 или SNMP) и N (для Sun XDR). Элемент *string-item* должен (на момент написания книги) принимать значение A*, a* или Z*. Для unpack длина строки извлекается из *length-item*, но если указать *, она игнорируется:

```
unpack "C/a", "\04Gurusamy";        # дает "Guru"
unpack "a3/A* A*", "007 Bond  J "; # дает (' Bond', 'J')
pack "n/a* w/a*","hello,","world"; # дает "\000\006hello,\005world"
```

Элемент *length-item* не возвращается unpack явно. Добавление *счетчика* к букве *length-item* вряд ли принесет какую-либо пользу, если только эта буква не A, a или Z. Упаковка с форматом *length-item*, равным a или Z, может вводить нулевые символы (\0), которые Perl не считает допустимыми в числовых строках.

За целочисленными форматами s, S, l и L может непосредственно следовать ! для обозначения родных для архитектуры коротких или длинных чисел вместо ровно 16 или 32 разрядов соответственно. Сегодня это представляет проблему в основном на 64-разрядных платформах, где родные для архитектуры короткие и длинные числа могут иначе представляться местными компиляторами C. (i! и I! тоже допустимы, но только для полноты картины; они идентичны i и I.)

Фактический размер родных для архитектуры типов short (короткий), int (целый), long (длинный) и long long (удвоенный длинный) в байтах на той платформе, где был скомпилирован Perl, тоже доступны посредством модуля Config:

```
use Config;
say $Config{shortsize};
say $Config{intsize};
say $Config{longsize};
say $Config{longlongsize};
```

Из того, что *Configure* известен размер long long, не обязательно следует, что вам доступны форматы q или Q. (В некоторых системах это так, но у вас, вероятно, не такая. Пока.)

Целочисленные форматы длиной более одного байта (s, S, i, I, l и L) по природе своей непереносимы между процессорами, поскольку подчиняются родному для архитектуры порядку следования байтов. Чтобы получить переносимые упакованные целые, используйте форматы n, N, v и V; для них порядок следования байтов и размеры известны.

Числа с десятичной запятой имеют только родной, аппаратный формат. Из-за многообразия форматов с десятичной запятой и отсутствия стандартного «сетевого» их представления, надежный обмен этими данными невозможен. Это значит,

что упакованные числа с десятичной запятой, записанные на одной машине, могут не читаться на другой. Проблема не решается, даже если обе машины используют арифметику с десятичной запятой IEEE, поскольку порядок записи байтов в память (прямой и обратный) не входит в спецификацию IEEE.

Perl внутренне использует представление с двойной точностью во всех вычислениях с десятичной запятой, поэтому при преобразовании из двойной точности в обычную и обратно теряется точность представления. Это значит, что unpack("f", pack("f", $foo)) не всегда равно $foo.

Программист берет на себя ответственность за любые особенности выравнивания или форматирования, относящиеся к другим программам, особенно к созданным компилятором С – а это зверь со своими представлениями о том, как размещать структуру на С для конкретной рассматриваемой архитектуры. Для этого при упаковке придется добавить достаточное количество x. Например, объявление С:

```
struct foo {
    unsigned char c;
    float f;
};
```

можно вывести с использованием формата "C x f", "C x3 f" или даже "f C". Функции pack и unpack считают входные и выходные данные однообразными байтовыми последовательностями, поскольку не могут знать, куда данные отправляются или откуда поступают.

Применение модификатора ! к формату @ или определяет смещение в байтах внутри упакованных строк. Это может повысить эффективность, но при этом требует более полных знаний о строке и размерах других форматов.

Модификаторы < и >, определяющие порядок следования байтов, предназначены для использования с форматами d, D, f, F, i, I, j, J, l, L, p, P, q, Q, s и S. Эти модификаторы упаковывают целые числа с определенным порядком следования байтов. Модификатор < принудительно устанавливает обратный (little-endian) порядок следования байтов, а > – прямой (big-endian):

```
$out = pack 'L>', 0xDEADBEEF; # "\xDE\xAD\xBE\xEF"
$out = pack 'L<', 0xDEADBEEF; # "\xEF\xBE\xAD\xDE"
```

Дополнительные примеры

Рассмотрим еще несколько примеров. Первая пара упаковывает числовые значения в байты:

```
$out = pack "CCCC", 65, 66, 67, 68: # $out eq "ABCD"
$out = pack "C4", 65, 66, 67, 68;   # то же самое
```

Следующая инструкция делает то же самое с буквами Юникода, заключенными в кружок:

```
$foo = pack("U4",0x24b6,0x24b7,0x24b8,0x24b9);
```

Следующая инструкция делает то же самое, вводя пару пустых символов:

```
$out = pack "CCxxCC", 65, 66, 67, 68, # $out eq "AB\0\0CD"
```

Упаковка коротких чисел не значит, что достигнута переносимость:

```
$out = pack "s2", 1, 2; # "\1\0\2\0" с обратным порядком следования байтов
                        # "\0\1\0\2" с прямым порядком следования байтов
```

При упаковке двоичных и шестнадцатеричных чисел *счетчик* указывает количество битов или полубайтов, а не создаваемых байтов:

```
$out = pack "B32", "01010000011001010111001001101100",
$out = pack "H8", "5065726c";      # обе возвращают "Perl"
```

Длина поля a относится только к одной строке:

```
$out = pack "a4", "abcd", "x", "y", "z"; # "abcd"
```

Это ограничение можно обойти, указав несколько спецификаторов:

```
$out = pack "aaaa",  "abcd", "x", "y", "z"; # "axyz"
$out = pack "a" x 4, "abcd", "x", "y", "z"; # "axyz"
```

Формат a осуществляет заполнение нулями:

```
$out = pack "a14", "abcdefg"; # "abcdefg\0\0\0\0\0\0\0"
```

Следующий шаблон упаковывает запись C struct tm (по крайней мере, в некоторых системах):

```
$out = pack "i9pl", gmtime(), $tz, $toff;
```

Обычно тот же формат можно использовать с функцией unpack, хотя некоторые спецификаторы действуют иначе, особенно a, A и Z.

Если нужно объединить текстовые поля фиксированной ширины, применяйте pack с *TEMPLATE* из нескольких форматов A или a:

```
$string = pack("A10" x 10, @data);
```

Нельзя сказать, что «A» слишком опасен: он прекрасно справляется с внутренним представлением Юникода в Perl. Но этот спецификатор производит дополнение по кодам символов, а не по логическим позициям вывода. Если потребуется обработать ваше résumé, это можно сделать, как показано ниже:

```
pack("(A10)2", "re\x{301}sume\x{301}", "work")'
"résumé work          "
```

Но если потребуется повторить обработку, вы можете получить:

```
say pack("(A10)2", "resume", "work")
resume    work
```

В разделе «Графемы и нормализация» главы 6 демонстрируется, как с помощью метода columns из модуля Unicode::GCString реализовать корректное дополнение при наличии в строке управляющих символов, комбинационных знаков и широких (занимающих две позиции вывода) букв, которые можно найти во многих восточноазиатских алфавитах.

Если потребуется объединить текстовые поля переменной длины через разделитель, используйте функцию join:

```
$string = join(" and ", @data);
$string = join("", @data); # пустой разделитель
```

Во всех приведенных примерах шаблонами служили литеральные строки, однако нет причин, почему нельзя загружать шаблоны из файла на диске. На этой функции можно построить целую систему реляционной базы данных. (Не станем уточнять, как это охарактеризует автора подобной затеи.)

unpack

```
unpack TEMPLATE, EXPR
```

Эта функция оказывает действие, обратное pack: она распаковывает строку (*EXPR*), представляющую структуру данных, в список значений согласно шаблону *TEMPLATE* и возвращает эти значения. В скалярном контексте она может применяться для распаковки одного значения. Шаблон *TEMPLATE* имеет формат, сходный с используемым в функции pack, – он задает порядок следования и типы значений, которые должны быть распакованы. Подробное описание *TEMPLATE* приведено в описании функции pack. Недопустимый элемент в *TEMPLATE* либо попытка выйти за пределы строки в форматах x, X или @ вызывает исключение.

Строка разбивается на фрагменты, описываемые *TEMPLATE*. Каждый участок отдельно преобразуется в значение. Обычно байты в строке либо получены с помощью pack, либо представляют некую структуру на C.

Если счетчик повторений поля больше, чем позволяет остаток входной строки, счетчик автоматически уменьшается. (Обычно в таких случаях в качестве счетчика повторений следует использовать *.) Если входная строка длиннее, чем описано в *TEMPLATE*, оставшаяся часть строки игнорируется.

Функция unpack может быть полезна и при работе с обычными текстовыми файлами, не обязательно двоичными. Предположим, что имеется файл данных с такими записями:

```
2009 The Graveyard Book                     Neil Gaiman
2008 The Yiddish Policemen's Union          Michael Chabon
2007 Rainbows End                           Vernor Vinge
2006 Spin                                   Robert Charles Wilson
2005 Jonathan Strange & Mr Norrell          Susanna Clarke
2004 Paladin of Souls                       Lois McMaster Bujold
2003 Hominids                               Robert J. Sawyer
2002 American Gods                          Neil Gaiman
2001 Harry Potter and the Goblet of Fire    J. K. Rowling
```

Такой файл мог быть создан с помощью функции printf, описанной выше в этой главе, или с применением форматов, описанных в следующем разделе. Он так же мог быть создан какими-либо внешними инструментами. В любом случае, не получится применить split для выделения полей, поскольку границы полей определяются не особым разделителем, а смещениями в записи. Поэтому, хотя это обычные текстовые записи, фиксированный формат позволяет разобрать их с помощью unpack:

```
use v5.14;
while (<>) {
    my($year, $title, $author) = unpack("A4 x A39 A*", $_);
    say "$author won ${year}'s Hugo for $title.";
}
```

(Мы написали ${year}'s, потому что Perl посчитал бы $year's за $year::s. Если бы в исходном коде программы использовалась директива use utf8, включающая поддержку кодировки UTF-8, можно было бы безопасно использовать форму записи $year's.)

Кроме полей, допустимых в pack, можно задавать для поля префикс %*число*, что создает простую контрольную сумму объекта с указанным количеством разрядов, а не сам объект. По умолчанию это 16-разрядная сумма. Контрольная сумма создается путем сложения числовых значений распакованных величин (для строковых полей берется сумма ord($char), а для битовых полей – сумма нулей и единиц). Например, следующий код рассчитывает то же число, что и программа SysV *sum*(1):

```
undef $/;
$checksum = unpack ("%32C*", <>) % 65535;
```

Следующая строка подсчитывает число установленных в единицу разрядов в битовой строке:

```
$setbits = unpack "%32b*", $selectmask;
```

Вот простой декодер BASE64:

```
while (<>) {
    tr#A-Za-z0-9+/##cd;                  # удалить символы не-base64
    tr#A-Za-z0-9+/# -_#;                 # преобразовать в формат uuencode
    $len = pack("c", 32 + 0.75*length);  # вычислить байт длины
    print unpack("u", $len . $_);        # uudecode и вывод
}
```

Форматы p и P должны использоваться с большой осторожностью. Поскольку Perl не имеет механизма проверки допустимости адреса в памяти, переданного функции unpack и хранящего значение, передача недопустимого указателя наверняка приведет к пагубным последствиям.

Если шаблон содержит больше спецификаторов, чем полей во входной строке, или если счетчики повторений приводят к выходу за границы строки, сложно предсказать, какой результат получится: счетчик может быть уменьшен; или unpack может произвести пустые строки или нули, или возбудить исключение. Если входная строка длиннее, чем описывает шаблон *TEMPLATE*, остаток входной строки игнорируется.

Форматы шаблонов

В Perl имеется механизм, позволяющий создавать простые отчеты и диаграммы, которые вы часто могли видеть выползающими из матричного принтера. (Как! У вас нет ничего подобного?) Для упрощения этой задачи Perl помогает запрограммировать вывод страницы близко к тому, как она будет выглядеть при печати. Он позволяет следить, например, за количеством строчек на странице, номером текущей страницы, управлять выводом заголовков страницы и т.д. Ключевые слова заимствованы из FORTRAN: format – для объявления и write – для выполнения; см. соответствующие разделы главы 27. К счастью, структура читается значительно легче – скорее как в инструкции PRINT USING языка BASIC. Можете считать, что это *nroff*(1) для бедных. (Если вы знакомы с *nroff*, это может прозвучать как антирекомендация.)

Форматы, как пакеты и подпрограммы, не выполняются, а объявляются, поэтому они могут находиться в любом месте программы. (Обычно все же лучше держать их все в одном месте.) У них есть собственное пространство имен, отличное от пространств других типов в Perl. Это значит, что если у нас есть функция с именем Foo, то это не то же самое, что формат Foo. Однако именем по умолчанию для формата, ассоциированного с некоторым дескриптором файла, служит имя этого дескриптора файла. Поэтому формат по умолчанию для STDOUT называется "STDOUT", а формат по умолчанию для дескриптора файла TEMP называется "TEMP". Они только выглядят одинаково, но не являются одним и тем же.

Форматы выходных записей объявляются так:

```
format NAME =
FORMLIST
.
```

Если имя *NAME* опущено, мы создаем формат STDOUT. *FORMLIST* состоит из последовательности строк, каждая из которых может иметь один из трех типов:

- Комментарий (строка начинается символом #).
- Строка шаблона (picture line), определяющая формат одной выводимой строки.
- Строка с аргументами, дающими значения, встраиваемые в предшествующую строку шаблона.

Строки шаблонов выводятся точно так, как они выглядят, за исключением того, что вместо некоторых полей подставляются их значения.[1] Каждое поле подстановки в строке шаблона начинается с @ (at) или ^ (крышка). Эти строки не подвергаются никакой интерполяции переменных. Поле @ (не путать с маркером массива @) представляет собой поле обычного типа; поле типа ^ предназначено для элементарного заполнения многострочных блоков текста. Размер поля задается заполнением его символами <, > или | для установки выравнивания соответственно по левому краю, правому краю или по центру. Если переменная превосходит заданную для нее ширину, она усекается.

Помните, что все эти разговоры о ширине и выравнивании разбиваются как волны о скалы, когда речь заходит о введении в шаблон «интересных» символов Юникода. Более того, проблемы начинаются даже с введением непечатаемых символов ASCII. Механизм поддержки форматов шаблонов предполагает, что каждый код символа соответствует ровно одной позиции вывода. В Юникоде это не всегда верно, поскольку в Юникоде многие коды не имеют отдельной позицию при выводе, а некоторые могут занимать две позиции. В разделе «Графемы и нормализация» главы 6 демонстрируется, как с помощью метода columns из модуля Unicode::GCString получить истинную ширину строки в позициях вывода.

В качестве альтернативной формы выравнивания чисел по правому краю можно использовать символы # (после начального @ или ^). Вместо одного из символов # можно вставить . (точку) и выровнять десятичные запятые. Если значение, пере-

---

[1] Даже эти поля поддерживают целостность тех колонок, в которые они вставляются. В строке нет ничего, что заставило бы поля расширяться, сжиматься или сдвигаться в том или ином направлении. Колонки, которые мы видим, священны в смысле WYSIWYG исходя из предположения, что используется моноширинный шрифт. Даже для управляющих символов предполагается наличие ширины в один символ.

даваемое какому-либо из этих полей, содержит символ перевода строки, выводится только текст перед этим символом. И, наконец, специальное поле @* может служить для вывода многострочных и неусеченных значений; в большинстве случаев оно должно быть единственным полем в строке шаблона.

Значения задаются в следующей строке в порядке следования полей в шаблоне. Выражения, служащие источником значений, разделяются запятыми. Все выражения вычисляются в списочном контексте перед обработкой строки, поэтому одно списочное выражение может породить несколько элементов списка. Выражения могут располагаться на нескольких строках, если заключены в фигурные скобки. (При этом открывающая фигурная скобка должна быть первой лексемой первой строки.) Это позволяет выравнивать значения по соответствующим полям формата для облегчения чтения.

Если выражение возвращает число со знаками после десятичной запятой и в соответствующем шаблоне указано, что дробная часть должна выводиться (т.е. в любом шаблоне, кроме множественных символов #, не содержащих точки .), то символ, выбранный в качестве десятичной запятой, определяется текущей установкой LC_NUMERIC. Это означает, например, что, если в среде этапа выполнения выбраны немецкие национальные установки, вместо точки будет использоваться запятая. Дополнительные сведения представлены на страницах руководства *perllocale*.

Пробельные символы \n, \t и \f внутри выражения считаются эквивалентными одному пробелу. То есть можно считать, что следующий фильтр применяется для каждого значения формата:

```
$value =~ tr/\n\t\f/ /
```

Оставшийся пробельный символ, \r, вызывает перевод строки, если строка шаблона это позволяет.

Поля шаблона, начинающиеся символом ^, а не @, обрабатываются особым образом. Поле # заполняется пробелами, если значение не определено. Для остальных типов полей крышка ^ включает режим заполнения. Передаваемое значение должно быть не выражением, а именем скалярной переменной, содержащей строку текста. Perl помещает в поле столько текста, сколько можно, а затем отсекает переднюю часть строки, так что при новом обращении к переменной может быть выведена следующая часть текста. (Да, это означает, что при выполнении вызова write переменная не сохраняется и сама изменяется. Сохранить исходное значение можно во временной переменной.) Обычно для вывода текстового блока используют ряд полей, выровненных по вертикали. Можно вывести в последнем поле символы «...», которые будут завершать выводимые строки, если текст слишком длинный, чтобы вывести его полностью. Изменить символы, по которым (или после которых) производится разрыв текста, можно, указав в переменной $: ($FORMAT_LINE_BREAK_CHARACTERS, если используется модуль English) список желаемых символов.

Следует понимать, что этот упрощенный порядок разрыва строк не имеет никакого отношения к более сложному порядку, который определяется приложением к стандарту Юникода «UAX #14: Unicode Line Breaking Algorithm». В случае с текстом Юникода должно использоваться свойство Line_Break=*VALUE* (коротко LB) каждого символа в соединении с замысловатыми таблицами, чтобы определить,

где разрешаются разрывы. Чтобы дать вам представление, насколько это сложно, ниже приводятся возможные значения *VALUE* свойства в \p{LB=*VALUE*}:

```
Ambiguous         Contingent_Break    Ideographic        Postfix_Numeric
Alphabetic        Close_Punctuation   Inseparable        Prefix_Numeric
Break_Both        Close_Parenthesis   Infix_Numeric      Quotation
Break_After       Combining_Mark      Line_Feed          Space
Break_Before      Complex_Context     Next_Line          Unknown
Mandatory_Break   Exclamation         Nonstarter         Word_Joiner
Break_Symbols     Glue                Numeric            ZWSpace
Carriage_Return   Hyphen              Open_Punctuation
```

Системы письменности, где не используются пробелы или дефисы, особенно чувствительны к разрывам строк, поэтому данный способ – единственно верный. Модуль Unicode::LineBreak из CPAN, включающий популярный модуль Unicode::GCString, полностью реализует требования приложения «UAX #14», включая обработку восточноазиатских систем письменности. Возможно, вам потребуется задействовать этот модуль при необходимости использовать более сложные наборы символов, чем ASCII.

В случае применения полей ^ могут создаваться записи переменной длины. Если форматируемый текст короткий, просто повторите строку формата с полем ^ несколько раз. Для коротких данных это приведет к выводу нескольких пустых строк. Для подавления вывода пустых строк поместите в любом месте строки символ ~ (тильда). (Сама тильда при выводе будет переведена в пробел.) Если рядом с первой тильдой поместить вторую, строка будет повторяться, пока не исчерпается весь текст в полях этой строки. (Это возможно потому, что поля ^ «съедают» выводимые ими строки. Но если используется поле типа @ вместе с двумя тильдами, то желательно, чтобы передаваемое в него выражение не выводило одно и то же значение до бесконечности! В этом случае используйте оператор shift или какой-либо другой с побочным эффектом, позволяющим исчерпать список значений.)

Обработка начала страницы по умолчанию осуществляется форматом, имя которого состоит из имени текущего дескриптора файла с присоединенными в конце символами _TOP. Она выполняется в начале вывода каждой страницы. См. описание write в главе 27.

Вот несколько примеров:

```
# отчет по файлу /etc/passwd
format STDOUT_TOP =
                        Файл Passwd
Имя                  Логин    Офис     Uid     Gid Каталог
-------------------------------------------------------------------
.
format STDOUT =
@<<<<<<<<<<<<<<<<<< @||||||| @<<<<<<@>>>> @>>>> @<<<<<<<<<<<<<<<<<
$name,               $login,  $office,$uid,$gid, $home
.

# отчет по бланку сообщения об ошибке
format STDOUT_TOP =
                        Сообщение об ошибке
```

```
@<<<<<<<<<<<<<<<<<<<<<<<     @|||        @>>>>>>>>>>>>>>>>>>>>>>>
$system,                      $%,        $date
-----------------------------------------------------------------
.
format STDOUT =
Сообщение: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
           $subject
Индекс: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<  ^<<<<<<<<<<<<<<<<<<<<<<<<<<<
        $index,                         $description
Приоритет: @<<<<<<<<<< Дата: @<<<<<<<  ^<<<<<<<<<<<<<<<<<<<<<<<<<<<
           $priority,        $date,    $description
От: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
    $from,                              $description
Предназначено: @<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<
               $programmer,             $description
~                                       ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                        $description
~                                       ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                        $description
~                                       ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                        $description
~                                       ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                        $description
~                                       ^<<<<<<<<<<<<<<<<<<<<<<<<...
                                        $description
```

Лексические переменные не видны в формате, если только объявление формата не находится в области видимости лексической переменной.

Можно чередовать применение `print` и `write` в одном и том же канале вывода, но придется самостоятельно обрабатывать специальную переменную `$-` (`$FORMAT_LINES_LEFT`, если используется модуль `English`).

Переменные форматов

Имя текущего формата хранится в переменной `$~` (`$FORMAT_NAME`), а имя текущего формата начала страницы – в переменной `$^` (`$FORMAT_TOP_NAME`). Номер текущей выводимой страницы хранится в `$%` (`$FORMAT_PAGE_NUMBER`), а число строк на странице – в `$=` (`$FORMAT_LINES_PER_PAGE`). Необходимость автоматической очистки буфера этого дескриптора определяется переменной `$|` (`$OUTPUT_AUTOFLUSH`). Строка, которую нужно выводить перед началом каждой страницы (кроме первой), хранится в `$^L` (`$FORMAT_FORMFEED`). Эти переменные устанавливаются отдельно для каждого дескриптора файла, поэтому, чтобы изменить переменные его формата, нужно сначала выбрать дескриптор файла с помощью `select`:

```
select((select(OUTF),
        $~ = "My_Other_Format"
        $^ = "My_Top_Format"
       )[0]);
```

Довольно безобразно, не так ли? Однако это общепринятая идиома, поэтому не удивляйтесь, увидев ее. Можно использовать временную переменную для хранения предыдущего дескриптора файла:

```
$ofh = select(OUTF);
$- = "My_Other_Format";
$^ = "My_Top_Format";
select($ofh);
```

В целом такой подход гораздо лучше, поскольку он не только улучшает читаемость, но и создает промежуточную инструкцию в коде, на которой происходит остановка при отладке в пошаговом режиме. При использовании модуля English имена переменных даже становятся понятными:

```
use English;
$ofh = select(OUTF);
$FORMAT_NAME     = "My_Other_Format"
$FORMAT_TOP_NAME = "My_Top_Format";
select($ofh);
```

Но курьезные вызовы select все же никуда не исчезли. Чтобы избежать их, воспользуйтесь модулем IO::Handle, поставляемым с Perl. В этом случае обращаться к специальным переменным можно с помощью методов, в именах которых используются буквы нижнего регистра:

```
use IO::Handle;
OUTF->format_name("My_Other_Format");
OUTF->format_top_name("My_Top_Format");
```

Так гораздо лучше!

Поскольку в строке значений, следующей за строкой шаблона, могут содержаться произвольные выражения (для полей типа @, но не полей ^), можно передать более сложную обработку другим функциям, скажем, sprintf или своим собственным. Например, чтобы вставить в число запятые:

```
format Ident =
    @<<<<<<<<<<<<<<<
    commify($n)
.
```

А чтобы вставить в поле фактические символы @, ~ или ^, сделайте так:

```
format Ident =
Я поместил @ сюда
          "@"
.
```

Для выравнивания по центру всей строки текста надо сделать примерно нижеследующее:

```
format Ident =
@||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
                                   "Какой-то текст"
.
```

Индикатор длины поля > обеспечивает выравнивание текста по правому краю поля, но само поле располагается в точности там, где это указано. Нет прямого способа: «переместить поле к правому краю страницы, какой бы широкой она ни была». Необходимо указать, где оно должно находиться относительно левого края. Те, кому это крайне необходимо, могут создать формат динамически, задав действительное число колонок, а затем выполнить его через eval:

```
$format = "format STDOUT = \n"
        . '^' . "<" x $cols . "\n"
        . '$entry' . "\n"
        . "\t^" . "<" x ($cols-8) . "--\n"
        . '$entry' . "\n"
        . ".\n";
print $format if $Debugging;
eval $format;
die $@ if $@;
```

Самой важной строкой здесь, вероятно, является `print`. **Результат вывода** `print` **может выглядеть следующим образом:**

```
format STDOUT =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$entry
    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<--
$entry
.
```

Вот маленькая программа, которая работает подобно утилите UNIX *fmt*(1):

```
format =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ~~
$_

.

$/ = "";
while (<>) {
    s/\s*\n\s*/ /g;
    write;
}
```

Нижние колонтитулы

В то время как имя текущего формата заголовка хранится в переменной `$^` **(**`$FORMAT_TOP_NAME`**), соответствующего механизма для нижнего колонтитула нет. Одна из главных проблем заключается в том, что размер формата не известен до тех пор, пока он не обработан. Эта задача – в нашем списке TODO.**[1]

Можно предложить такую стратегию: если нижний колонтитул имеет фиксированный размер, то выводить его, проверяя число оставшихся в формате строк `$-` **(**`$FORMAT_LINES_LEFT`**) перед каждой операцией** `write`**, и при необходимости самостоятельно выводить нижний колонтитул.**

Вот другая стратегия: откройте канал с помощью `open(MESELF, "|-")` **(см. описание** `open` **в главе 27) и всегда выполняйте** `write` **в** `MESELF` **вместо** `STDOUT`**. Затем пусть ваш дочерний процесс обработает свой** `STDIN` **и переставит верхние и нижние колонтитулы так, как вам хочется. Не очень удобно, но реализуемо.**

---

[1] Что, конечно, не гарантирует, что мы когда-нибудь это сделаем. Форматы являются отчасти пройденным этапом в наш век WWW, Юникода, XML, XSLT и неизбежно надвигающегося будущего.

Низкоуровневый доступ к форматированию

Для низкоуровневого доступа к механизму форматирования можно использовать встроенный оператор `formline` **и обратиться к** `$^A` **(переменная** `$ACCUMULATOR`**) непосредственно. (Форматы, в сущности, компилируются в последовательность вызовов** `formline`**.) Например:**

```
$str = formline <<'END', 1,2,3;
@<<<  @|||  @>>>
END

say "Вау, я запомнил '$^A' в аккумулятор!"
```

Чтобы создать подпрограмму `swrite` **(парную для** `write`**, как** `sprintf` **для** `printf`**), сделайте следующее:**

```
use Carp;
sub swrite {
    croak "Порядок использования: swrite PICTURE ARGS" unless @_;
    my $format = shift;
    $^A = "";
    formline($format, @_);
    return $^A;
}

$string = swrite(<<'END', 1, 2, 3);
Check me out
@<<<  @|||  @>>>
END
print $string;
```

Если используется модуль `IO::Handle`**, с помощью** `formline` **можно организовать перенос блоков текста по колонке 72:**

```
use FileHandle;
STDOUT->formline("^" . ("<" x 72) . "~~\n", $long_text);
```

А теперь приготовьтесь к изучению *большой* главы...

27

Функции

В этой главе описываются встроенные функции Perl в алфавитном порядке,[1] обеспечивающем удобство доступа к справочной информации. Описание каждой функции начинается краткой сводкой ее синтаксиса. Имена параметров, такие как *THIS*, представляют фактические выражения, а текст, следующий за синтаксической сводкой, описывает семантику применения фактических аргументов.

Функции, наряду с литералами и переменными, можно считать термами выражения либо префиксными операторами, которые обрабатывают аргументы, следующие за ними. В половине случаев мы функции так и называем – операторами.

Некоторые операторы (т. е. функции) принимают в качестве аргумента *LIST* (список). Элементы списка *LIST* должны разделяться запятыми (или оператором =>, причудливым вариантом запятой). Элементы списка вычисляются в списочном контексте, поэтому каждый элемент возвращает скалярное или списочное значение, в зависимости от своей чувствительности к списочному контексту. Каждое вычисленное значение, будь оно скаляром или списком, интерполируется как часть общей последовательности скалярных значений. Проще говоря, все списки объединяются в один плоский список. С точки зрения функции, получающей аргументы, общий аргумент *LIST* всегда является одномерным списком. (Чтобы интерполировать массив как один элемент, необходимо явно создать и интерполировать ссылку на массив.)

Аргументы предопределенных функций Perl можно заключать или не заключать в круглые скобки, но в синтаксических сводках данной главы скобки опущены. При использовании скобок применяется простое правило, вызывающее, время от времени, удивление: если нечто выглядит как функция, *это и есть* функция, поэтому приоритет не имеет значения. В противном случае это списочный или унарный оператор, и приоритет имеет значение. Будьте осторожны, поскольку даже

---

[1] Иногда тесно связанные между собой функции объединены на страницах руководства, и мы придерживаемся здесь такой же группировки. Например, чтобы найти описание `endpwent`, придется смотреть `getpwent`.

если между ключевым словом и левой скобкой поместить пробельный символ, такая конструкция все равно останется вызовом функции:

```
print 1+2*4;      # выведет 9.
print(1+2) * 4;  # выведет 3!
print (1+2)*4.   # Тоже выведет 3!
print +(1+2)*4;  # выведет 12.
print ((1+2)*4); # выведет 12.
```

Если запустить Perl с ключом *-w*, подобный код приведет к предупреждению. Например, вторая и третья строки из примера выше порождают такие сообщения:

```
print (...) interpreted as function at - line 2
Useless use of integer multiplication in void context at - line 2.
[print (...) интерпретируется как функция - в строке 2
 Бессмысленное использование умножения целых в пустом контексте - в строке 2]
```

Учитывая простоту определения некоторых функций, программисту предоставляется значительная свобода в способе передачи аргументов. Например, самый распространенный способ вызова chmod состоит в передаче прав доступа к файлу (т.е. режима) в первом аргументе:

```
chmod 0644, @array;
```

но в определении chmod просто сказано:

```
chmod LIST
```

поэтому с таким же успехом можно сказать:

```
unshift @array, 0644;
chmod @array;
```

Если первый аргумент списка не является допустимым режимом, вызов chmod завершается неудачей, но это – семантическая проблема этапа выполнения, не связанная с синтаксисом вызова. Если семантика требует передачи каких-либо специальных аргументов во главе списка, эти ограничения будут описаны в тексте.

В противоположность простым функциям, принимающим список *LIST*, другие функции налагают дополнительные синтаксические ограничения. Например, для push синтаксически правильным является такой вызов:

```
push ARRAY, LIST
```

Это означает, что push ожидает получить действительный массив в первом аргументе, но безразлична к следующим аргументам. Вот что означает *LIST* в конце. (Списки всегда идут в конце, так как они «съедают» все оставшиеся значения.) Если в синтаксической сводке есть аргументы, предшествующие *LIST*, они синтаксически различаются компилятором, а не просто семантически различаются интерпретатором при его последующем выполнении. Такие аргументы никогда не вычисляются в списочном контексте. Они могут вычисляться в скалярном контексте или быть особыми ссылочными аргументами, как массив для вызова push. (В описаниях функций мы рассказываем, где что.)

Для операций, основанных непосредственно на функциях библиотеки C, мы не стали дублировать системную документацию. Если в описании function предлагается обратиться к *function*(2), следует найти C-версию этой функции, чтобы больше узнать о ее семантике. Число в скобках указывает на раздел в руководстве

системного программиста, содержащий нужную страницу руководства, если в системе установлено руководство (man pages). (Или не содержащий, если руководство не установлено.)

Эти страницы руководства могут описывать системо-зависимые функции – такие, например, как теневые файлы паролей, списки управления доступом и т.д. Многие функции Perl, являющиеся производными от библиотечных функций C в UNIX, эмулируются даже на других платформах. Например, операционная система может не поддерживать системные вызовы *flock*(2) или *fork*(2); тем не менее, Perl постарается эмулировать их с использованием тех средств, которые предоставляет данная платформа.

Иногда оказывается, что документированная функция C принимает больше аргументов, чем соответствующая функция Perl. Обычно отсутствующие аргументы – это то, что Perl уже известно, например, длина предыдущего аргумента, поэтому передавать такие аргументы в Perl не требуется. Все остальные несоответствия обусловлены различиями в представлении дескрипторов файлов и значений кодов завершения в Perl и C.

В общем случае функции Perl, служащие обертками одноименных системных вызовов (например, *chown*(2), *fork*(2), *closedir*(2) и т.д.), возвращают истинное значение при успехе и `undef` в противном случае, на что указывают описания этой главы. В этом – их отличие от соответствующих интерфейсов библиотеки C, которые всегда возвращают `-1` при неудаче. Исключениями из этого правила являются `wait`, `waitpid` и `syscall`. Системные вызовы при неудаче также устанавливают специальную переменную `$!` (`$OS_ERROR`). Прочие функции этого не делают, разве что случайно.

Для функций, которые могут использоваться как в скалярном, так и в списочном контекстах, неудача в скалярном контексте часто обозначается ложным значением (обычно `undef`), а в списочном контексте – пустым списком. На успешное выполнение обычно указывает возвращаемое значение, которое вычисляется как истинное (в контексте).

Запомните следующее правило: *нет* правила, связывающего характер поведения функции в списочном контексте с ее поведением в скалярном контексте, и наоборот. Это поведение может значительно различаться.

Каждая функция знает, в каком вызвана контексте. Одна и та же функция возвращает список в списочном контексте, и значение наиболее подходящего типа – в скалярном контексте. Некоторые функции в скалярном контексте возвращают длину списка, который вернули бы в списочном контексте. Некоторые операторы возвращают первое значение в списке. Некоторые функции возвращают последнее значение в списке. Какие-то функции возвращают «другое» значение, когда нечто можно искать либо по номеру, либо по имени. Некоторые функции возвращают число успешных операций. В целом функции Perl делают в точности то, что вам нужно, если только вам не нужно единообразие.

Одно заключительное замечание: мы постарались быть последовательными в использовании терминов «байт» и «символ». Исторически эти термины всегда смешивались. Когда мы говорим «байт», то всегда подразумеваем восьмибитный символ. Когда мы говорим «символ», то подразумеваем абстрактный код Юникода. Это то, что программисты на C обычно хранили в своих переменных типа `char`, пока их размера хватало. В наши дни новым типом `char` стал тип `int`. Код символа Юникода –

это *видимый программисту символ*, неотрицательное целое число, соответствующее единственной сущности Юникода, иногда неформально называемой символом.

В настоящее время лишь несколько функций, не входящих в библиотеку поддержки регулярных выражений, имеют непосредственное отношение к графемам, но они образуют следующий уровень абстракции над кодами символов. Они являются *видимыми пользователю символами*, и могут, в свою очередь, включать символы, видимые программистами. Последовательность CR+LF является примером графемы, состоящей из двух кодов. Другим хорошим примером является графема ȭ, которая в разных ситуациях может занимать от 1 до 3 кодов, в зависимости от нормализации: \x{22D} в NFC, \x{6F}\x{303}\x{304} в NFD или \x{F5}\x{304} без нормализации. Поэтому, когда в этой главе мы говорим о символах, на самом деле мы подразумеваем коды символов, а если говорим о байтах, подразумеваем обычные, *не декодируемые* порядковые значения от 0 до 255.

Функции Perl по категориям

Вот функции Perl и ключевые слова, похожие на функции, разбитые на категории. Некоторые функции входят в несколько разделов.

Обработка скаляров

chomp, chop, chr, crypt, fc, hex, index, lc, lcfirst, length, oct, ord, pack, q//, qq//, reverse, rindex, sprintf, substr, tr///, uc, ucfirst, y///

Регулярные выражения и поиск по шаблону

m//, pos, qr//, quotemeta, s///, split, study

Числовые функции

abs, atan2, cos, exp, hex, int, log, oct, rand, sin, sqrt, srand

Обработка массивов

pop, push, shift, splice, unshift

В Perl версии v5.12 появилась возможность применять функции each, keys и values к массивам, если потребуется.

Обработка списков

grep, join, map, qw//, reverse, sort, unpack

Обработка хешей

delete, each, exists, keys, values

Ввод и вывод

binmode, close, closedir, dbmclose, dbmopen, die, eof, fileno, flock, format, getc, print, printf, read, readdir, readpipe, rewinddir, say, seek, seekdir, select (готовые дескрипторы файлов), syscall, sysread, sysseek, syswrite, tell, telldir, truncate, warn, write

Данные фиксированной длины и записи

pack, read, syscall, sysread, sysseek, syswrite, unpack, vec

Дескрипторы файлов, файлы и каталоги

-X, chdir, chmod, chown, chroot, fcntl, glob, ioctl, link, lstat, mkdir, open, opendir, readlink, rename, rmdir, select (готовые дескрипторы файлов), select (дескриптор выходного файла), stat, symlink, sysopen, umask, unlink, utime

Управление последовательностью выполнения команд

caller, continue, die, do, dump, eval, exit, __FILE__, goto, last, __LINE__, next, __PACKAGE__, redo, return, sub, wantarray

Области видимости

caller, import, local, my, no, our, package, state, use

state **доступна, только если включена особенность** "state" **или если она вызывается с префиксом** CORE::. **См. описание** feature. **Эту функцию можно также включить в текущую область видимости директивой** use v5.10 **или с более высоким номером версии.**

switch

break, continue, default, given, when

За исключением continue, **служащего выражением, а не блоком, эти ключевые слова становятся доступны только после включения** "switch". **Тот же эффект дает включение в текущую область видимости директивы** use v5.10 **или с более высоким номером версии. См. раздел «Оператор given» в главе 4.**

Разное

defined, dump, eval, formline, lock, prototype, reset, scalar, undef, wantarray

Процессы и группы процессов

alarm, exec, fork, getpgrp, getppid, getpriority, kill, pipe, qx//, setpgrp, setpriority, sleep, system, times, wait, waitpid

Библиотечные модули

do, import, no, package, require, use

Классы и объекты

bless, dbmclose, dbmopen, package, ref, tie, tied, untie, use

Низкоуровневый доступ к сокетам

accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, send, setsockopt, shutdown, socket, socketpair

Межпроцессные взаимодействия в System V

msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmctl, shmget, shmread, shmwrite

Получение информации о пользователях и группах

endgrent, endhostent, endnetent, endpwent, getgrent, getgrgid, getgrnam, getlogin, getpwent, getpwnam, getpwuid, setgrent, setpwent

Получение информации о сети

endprotoent, endservent, gethostbyaddr, gethostbyname, gethostent, getnetbyaddr, getnetbyname, getnetent, getprotobyname, getprotobynumber, getprotoent, getservbyname, getservbyport, getservent, sethostent, setnetent, setprotoent, setservent

Время

gmtime, localtime, time, times

Функции, имеющие отношение к Юникоду

binmode, chomp, chop, chr, dbmopen, fc, getc, index, lc, lcfirst, length, m//, my, open, ord, our, pack, package, pos, print, printf, quotemeta, read, readline, reverse, rindex,

s///, seek, sort, split, sprintf, state, substr, sysopen, sysread, sysseek, syswrite, tell, tr///, truncate, uc, ucfirst, unpack, write, y///

Функции Perl в алфавитном порядке



Многие из последующих имен функций аннотированы значками. Вот их значение:

- $_ Использует $_ ($ARG) в качестве переменной по умолчанию.
- $! Устанавливает $! ($OS_ERROR) при ошибках системных вызовов.
- $@ Возбуждает исключение; используйте eval для перехвата $@ ($EVAL_ERROR).
- $? Устанавливает $? ($CHILD_ERROR) при выходе из порожденного процесса.
- T Делает возвращаемые данные мечеными.
- T Делает возвращаемые данные мечеными при некоторых системных, национальных настройках или настройках дескрипторов.
- X ARG Возбуждает исключение при получении аргумента недопустимого типа.
- X RO Возбуждает исключение при модификации объекта, доступного только для чтения.
- X T Возбуждает исключение при получении меченых данных.
- X U Возбуждает исключение, если не реализована на данной платформе.
- X WIDE Возбуждает исключение, если передается строка, содержащая символы с порядковыми значениями больше 255.

Функции, возвращающие меченые данные, получив меченые данные на входе, особо не выделены, поскольку так поступает большинство функций. В частности, любая функция, которая применяется к %ENV или @ARGV, возвращает меченые данные.

Функции, отмеченные значком X ARG, возбуждают исключение, когда ожидают получить, но не получают аргумент определенного типа (например, дескриптор файла для операций ввода/вывода, ссылки для bless и т.д.).

Функциям, отмеченным значком X RO, иногда требуется изменить свои аргументы. Если они не могут изменить свой аргумент, из-за того, что для него установлен атрибут «только для чтения», то возбуждают исключение. Примерами переменных «только для чтения» служат специальные переменные, содержащие данные, сохраненные при поиске по шаблону, и переменные, являющиеся псевдонимами констант.

Наличие функции со значком X U зависит от платформы. Хотя многие такие функции получили свои имена от функций из библиотеки C в UNIX, не следует считать, что пользователи других платформ не могут ими пользоваться. Многие из этих функций эмулируются, даже те, для которых мы этого не ожидаем: например fork в системах Win32. Дополнительную информацию о переносимости и специфических для системы функциях можно найти на странице руководства *perlport*, а также в специфической для платформы документации, поставляемой с версией Perl для конкретной платформы.

Функции, отмеченные значком X WIDE, возбуждают исключения, если получают недекодированные строки символов, слишком больших, чтобы уместиться в один байт.

Функции, возбуждающие прочие исключения, отмечаются значком $@. Это, в том числе, математические функции, возбуждающие ошибки диапазона аргументов, например sqrt(-1).

abs

$_

```
abs VALUE
abs
```

Возвращает абсолютное значение своего аргумента.

```
$diff = abs($first - $second)
```

Здесь и в последующих примерах хороший стиль программирования (и прагма strict) требует добавления модификатора my для объявления новой переменной с лексической областью видимости, например:

```
my $diff = abs($first - $second);
```

Однако для ясности мы опускаем my в большинстве примеров. Просто считайте, что все такие переменные были объявлены раньше, если это вам нравится.

accept




```
accept SOCKET, PROTOSOCKET
```

Эту функцию используют серверные процессы, которым необходимо ожидать подключений клиентов через сокеты. *PROTOSOCKET* – это дескриптор файла, уже открытый оператором socket и привязанный к одному из сетевых адресов сервера или к INADDR_ANY. Выполнение приостанавливается до создания соединения, при этом дескриптор файла *SOCKET* открывается и прикрепляется к вновь созданному соединению. Исходный *PROTOSOCKET* остается без изменений; его единственное назначение состоит в том, чтобы быть клонированным в действительный сокет. При успешном обращении функция возвращает адрес, с которым произведено соединение, и ложное значение в противном случае. Например:

```
unless ($peer = accept(SOCKET, PROTOSOCKET)) {
    die "Невозможно принять соединение: $!\n";
}
```

В системах, где это допустимо, будет установлен флаг закрытия при exec для вновь открытого дескриптора файла, если этого требует значение $^F ($SYSTEM_FD_MAX).

См. *accept*(2). См. также пример в разделе «Сокеты» главы 15.

alarm




```
alarm EXPR
alarm
```

Посылает текущему процессу сигнал SIGALRM по истечении *EXPR* секунд.

Одновременно может быть активен только один таймер. Каждое обращение к функции отключает предшествующий таймер. Значение *EXPR* равное 0 отключает пре-

дыдущий таймер без запуска нового. Функция возвращает остаток времени предыдущего таймера.

```
print "Ответь мне в течение минуты или умри: ";
alarm(60);            # завершить программу через минуту
$answer = <STDIN>;
$timeleft = alarm(0); # сбросить таймер
say "У вас осталось $timeleft секунд";
```

Смешивание вызовов alarm **и** sleep **обычно является ошибкой, потому что во многих системах для реализации *sleep*(3) используется механизм системного вызова *alarm*(2). В более старых системах истекшее время может быть на одну секунду меньше заданного из-за способа подсчета секунд. Кроме того, сильно загруженная система может не сразу приступить к выполнению процесса. См. в главе 15 сведения об обработке сигналов.**

Если требуется таймер с более высокой точностью, чем одна секунда, можно воспользоваться модулем Time::HiRes**. Многоопытные программисты могут использовать версию** select **с четырьмя аргументами (оставив три первых аргумента неопределенными), или функцию** syscall **для обращения к *setitimer*(2), если система это поддерживает.**

atan2

```
atan2 Y, X
```

Возвращает главное значение арктангенса *Y/X* **в диапазоне от** $-\pi$ **до** π**. Быстро получить приближенное значение** π **можно так:**

```
$pi = atan2(1,1) * 4;
```

Для вычисления тангенсов можно обратиться к функции tan **из модуля** Math::Trig **или** POSIX **либо просто использовать известное соотношение:**

```
sub tan { sin($_[0]) / cos($_[0]) }
```

Если какой-либо один или оба аргумента равны 0, возвращаемое значение зависит от реализации; за дополнительной информацией обращайтесь к своей странице *atan2*(3) справочного руководства.

bind




```
bind SOCKET, NAME
```

Связывает адрес (имя) с уже открытым сокетом, заданным уже открытым дескриптором файла *SOCKET*. **Возвращает истинное значение в случае успеха и ложное в противном случае. Аргумент** *NAME* **должен быть упакованным адресом сокета, имеющим надлежащий тип.**

```
use Socket;
$port_number = 80;      # допустим, мы хотим быть веб-сервером
$sockaddr = sockaddr_in($port_number, INADDR_ANY);
bind SOCK, $sockaddr || die "Невозможно связать $port_number: $!";
```

См. *bind*(2). См. также примеры в разделе «Сокеты» главы 15. Обычно следует отдавать предпочтение высокоуровневому интерфейсу доступа к сокетам, предоставляемому стандартным модулем IO::Socket.

binmode




```
binmode FILEHANDLE, IOLAYER
binmode FILEHANDLE
```

Устанавливает для дескриптора файла *FILEHANDLE* семантику, заданную аргументом *IOLAYER*. Если аргумент *IOLAYER* опущен, применяется двоичная (или raw) семантика. Если *FILEHANDLE* представлен выражением, его значение принимается за имя дескриптора файла или ссылку на дескриптор файла, в зависимости от обстоятельств. В случае успеха функция возвращает истинное значение, в случае неудачи – ложное.

Функцию binmode нужно вызывать после open, но перед любыми операциями ввода/вывода с дескриптором файла. Единственный способ изменить режим дескриптора файла – открыть его повторно, поскольку разные уровни *IOLAYER* могут хранить разные фрагменты данных в разных буферах. Это ограничение в будущем может быть ослаблено.

В прежние времена функция binmode применялась в основном в операционных системах, библиотеки времени выполнения которых различали текстовые и двоичные файлы. В таких системах binmode служила для отключения текстовой семантики, использовавшейся по умолчанию. Однако с приходом Юникода все программы во всех системах должны каким-то образом проводить это различие.

В наше время есть только один тип двоичного файла (если говорить о Perl), но много типов текстовых файлов, которые тоже было бы желательно обрабатывать в Perl единым способом. Поэтому в Perl есть единственный внутренний формат представления текста Юникода: UTF-8.

Поскольку существует много типов текстовых файлов, при вводе часто требуется транслировать текстовые файлы в UTF-8, а при выводе в какой-либо из унаследованных наборов символов или в другое представление Юникода.

Посредством фильтров ввода/вывода можно сообщить Perl, как точно (или не точно) осуществлять такие переводы. Например, фильтр ":text" указывает на необходимость обработки текста вообще, без уточнения, какого рода текст обрабатывается.

Но такие фильтры ввода/вывода, как ":utf8" и ":encoding(Latin1)", указывают, в каком формате производить чтение и запись текста.

С другой стороны, фильтры ввода/вывода вроде ":raw" запрещают Perl трогать данные своими «шаловливыми ручками».

Подробнее о том, как работают (или будут работать) фильтры ввода/вывода, мы рассказываем в описании функции open. Оставшаяся часть данного изложения посвящена действиям binmode в отсутствие аргумента *IOLAYER*, т.е. историческому значению binmode, которое эквивалентно:

```
binmode FILEHANDLE, ":raw";
```

Если не указано другое, Perl предполагает, что вновь открытый файл нужно читать и писать в текстовом режиме. Текстовый режим означает, что внутренним указателем конца строки будет \n (перевод строки). Все системы используют \n в качестве внутреннего указателя конца строки, но его действительное представление различается от системы к системе, устройства к устройству и даже от файла к файлу, в зависимости от способа обращения к файлу. В старых системах

(включая MS-DOS и VMS) то, что программа видит как \n, может быть не тем, что физически хранится на диске. Операционная система способна сохранять текстовые файлы с последовательностями \cM\cJ, которые транслируются при вводе и представляются программе как \n, а \n при выводе из программы в файл транслируются в \cM\cJ. Функция binmode отключает эту автоматическую трансляцию в таких системах.

В отсутствие аргумента *IOLAYER* функция binmode не оказывает никакого действия в UNIX (включая Mac OS X), где \n служит для обозначения конца каждой строки и представляется одним символом. (Однако это могут быть разные символы: UNIX использует \cJ, а старые Маки – \cM. Все это не имеет значения.)

В следующем примере показано, как сценарий Perl может прочесть изображение в формате GIF из файла и вывести его на стандартное устройство вывода. В системах, которые переводят буквальные данные в нечто отличное от их точного физического представления, необходимо настроить оба дескриптора. Хотя можно использовать фильтр ":raw" напрямую при открытии GIF, не получится так просто сделать это для предварительно открытых дескрипторов файлов вроде STDOUT:

```
binmode(STDOUT, ":raw")
    || die "невозможно применить фильтр raw к STDOUT: $!";
open(GIF, "< :raw", "vim-power.gif")
    || die "невозможно открыть vim-power.gif: $!";
while (read(GIF, $buf, 1024)) { # теперь байты, а не символы
    print STDOUT $buf;
}
```

Обратите внимание, что при использовании встроенного фильтра UTF-8:

```
binmode(HANDLE, ":utf8");
```

если HANDLE является дескриптором входного файла, вы должны быть готовы вручную обрабатывать ошибки кодирования, потому что, начиная с версии v5.14, механизм по умолчанию слишком либерально относится к неправильным кодам UTF-8. Самый простой и лучший способ обработки ошибок кодирования состоит в том, чтобы вообще не пропускать их.

```
use warnings FATAL => "utf8";
```

Даже если вы задействуете модуль Encode, как показано ниже:

```
binmode(HANDLE, ":encoding(utf8)")
```

все равно следует сделать предупреждения UTF-8 фатальными, как показано выше, потому что иначе вы не будете получать исключения в случае появления ошибок. (Исключения всегда предпочтительнее искаженного текста. Такой текст все равно не сделает вас счастливыми.)

bless




```
bless REF, CLASSNAME
bless REF
```

Эта функция сообщает объекту ссылки, на который указывает ссылка *REF*, что с этих пор он является объектом в пакете *CLASSNAME* – или текущем пакете, если имя класса *CLASSNAME* не задано. Если *REF* не является действующей ссылкой, воз-

буждается исключение. Для удобства bless возвращает ссылку, так как часто это последний вызов в подпрограмме-конструкторе. Например:

```
$pet = Beast->new(TYPE => "cougar"  NAME => "Clyde");

# далее в Beast.pm:
sub new {
    my $class = shift;
    my %attrs = @_;
    my $self  = { %attrs };
    return bless($self, $class);
}
```

Как правило, следует «освящать» объекты в классы, имена которых имеют смешанный регистр. Пространства имен с именами в нижнем регистре зарезервированы для внутреннего использования в качестве прагм Perl (директив компилятора). Имена встроенных типов (например, SCALAR, ARRAY, HASH и т.д., не говоря уже о базовом классе всех классов UNIVERSAL) составлены из символов верхнего регистра, поэтому таких имен тоже желательно избегать.

Аргумент *CLASSNAME* не должен быть ложным значением; «освящение» в ложные пакеты не поддерживается и может приводить к непредсказуемым результатам.

Отсутствие парного оператора curse (проклятие) ошибкой не является; зато у нас есть оператор sin (грех). См. также главу 12, где подробнее рассказано об освящении (и благодеяниях) объектов.

break

```
break
```

Осуществляет преждевременный (до окончания предложения when) выход из блока given. Поддержка этого ключевого слова включается особенностью switch; см. описание прагмы feature в главе 29.

caller

```
caller EXPR
caller
```

Возвращает сведения о стеке вызовов текущей подпрограммы и сопутствующую информацию. При вызове без аргумента возвращает имя пакета, имя файла и номер строки, из которой была вызвана подпрограмма, выполняемая в данный момент:

```
($package, $filename, $line) = caller;
```

Вот пример исключительно привередливой функции, использующей специальные лексемы __PACKAGE__ и __FILE__, описываемые в главе 2:

```
sub careful {
    my ($package, $filename) = caller;
    unless ($package eq __PACKAGE__ && $filename eq __FILE__) {
        die "Ты не должен был вызывать меня, $package!\n";
    }
    say "свободно вызвал меня";
```

```
}

sub safecall {
    careful();
}
```

При вызове с аргументом функция caller интерпретирует *EXPR* как количество кадров стека, на которое надо вернуться назад от текущего. Например, аргумент 0 означает текущий кадр стека, 1 означает кадр стека вызвавшей подпрограммы, 2 означает кадр стека подпрограммы, вызвавшей вызвавшую, и т.д. Функция возвращает также дополнительную информацию, как показано ниже:

```
my $i = 0;
while (my ($package, $filename, $line, $subroutine,
        $hasargs, $wantarray, $evaltext, $is_require,
        $hints, $bitmask, $hinthash) = caller($i++) )
{
    ...
}
```

Если кадр стека представляет вызов подпрограммы, $hasargs принимает истинное значение, если имеется собственный массив @_ (не заимствованный у вызвавшей подпрограммы). В противном случае $subroutine может иметь значение "(eval)", если кадр является не вызовом подпрограммы, а eval. В этом случае устанавливаются дополнительные элементы $evaltext и $is_require: $is_require принимает истинное значение, если кадр создан командой require или use, а $evaltext содержит текст команды eval *EXPR*. В частности, для команды eval *BLOCK* элемент $filename есть "(eval)", а $evaltext не определена. (Обратите также внимание, что каждая инструкция use создает кадр require внутри кадра eval *EXPR*.) Элементы $hints, $bitmask и $hinthash представляют собой внутренние значения: пожалуйста, используйте их, только если принадлежите к элите чудотворцев.[1]

В качестве еще более сильного волшебства caller также заполняет массив @DB::args аргументами, переданными в данном кадре стека, но только при вызове из пакета DB. См. главу 18.

Имейте в виду, что оптимизатор может оптимизировать кадры стека вызовов еще до того, как caller получит шанс извлечь информацию. Это означает, что вызов caller(*N*) может не вернуть информацию о кадрах стека вызовов для *N* > 1. В частности, @DB::args может хранить информацию, полученную предыдущим вызовом caller.

Также следует понимать, что установка @DB::args – это *лишь попытка без гарантий*; это действие возможно использовать для отладки или вывода трассировочной информации, но не следует полагаться на информацию, хранящуюся в @DB::args. В частности, @_ содержит синонимы для текущего массива @_ вызывающей подпрограммы. Perl не создает копии @_ при входе в подпрограмму, поэтому @DB::args будет отражать любые изменения, которые подпрограмма внесет в @_ после вызова. Кроме того, @DB::args, подобно @_, не хранит явные ссылки на

---

[1] $hinthash – это ссылка на хеш, содержащий значение %^H на момент компиляции вызывающего кода, или undef, если %^H была пустой. Не изменяйте значения в этом хеше, так как они являются фактическими значениями, хранящимися в дереве операций.

свои элементы, потому в некоторых случаях его элементы могут быть освобождены, а занимаемая ими память может быть отведена под другие переменные или временные значения. Наконец, текущая реализация имеет побочный эффект, который выражается в том, что отменить можно только `shift @_` (но не `pop` или `splice`), а если была получена ссылка на `@_`, вы наверняка получите щелчок по носу. То есть в действительности `@DB::args` является гибридом текущих и начальных значений в `@_`. Будьте осторожны.

chdir

```
chdir EXPR
chdir
```

Изменяет текущий каталог процесса на *EXPR*, если это возможно. Если аргумент *EXPR* опущен, подразумевается исходный каталог вызвавшего процесса. Функция возвращает истинное значение в случае успеха и ложное в противном случае.

```
chdir "$prefix/lib" || die "Невозможно cd в $prefix/lib: $!\n"
```

См. также модуль `Cwd`, который позволяет автоматически отслеживать текущий каталог.

В системах, поддерживающих *fchdir*(2), в качестве *EXPR* можно использовать дескриптор файла или каталога. В системах, не поддерживающих *fchdir*(2), передача дескриптора вызовет исключение во время выполнения.

chmod



```
chmod LIST
```

Изменяет права доступа для списка файлов. Первым элементом списка должен быть режим доступа в числовом выражении, как в системном вызове *chmod*(2). Функция возвращает количество успешно измененных файлов. Например, вызов

```
$cnt = chmod 0755, "file1", "file2";
```

установит `$cnt` в 0, 1 или 2, в зависимости от того, сколько файлов было изменено. Успех определяется отсутствием ошибки, а не фактическим изменением, поскольку файл мог иметь тот же режим до выполнения операции. Ошибка может означать, что у пользователя недостаточно прав на изменение режима, поскольку он не является владельцем файла или суперпользователем. Проверьте `$!`, чтобы выяснить фактическую причину неудачи.

Вот более типичное применение:

```
chmod(0755, @executables) == @executables
        ||die "невозможно выполнить chmod для некоторых @executables: $!";
```

Чтобы узнать, какие файлы не удалось изменить, сделайте что-нибудь в таком роде:

```
@cannot = grep {not chmod 0755, $_} "file1", "file2", "file3";
die "$0: невозможно выполнить chmod для @cannot\n" if @cannot;
```

В этой идиоме используется функция `grep` для отбора только тех элементов списка, для которых не удалось выполнить `chmod`.

В системах, поддерживающих *fchmod*(2), в списке аргументов можно также передавать дескрипторы файлов. В системах, не поддерживающих *fchmod*(2), передача дескриптора файла вызовет исключение во время выполнения. Для корректной обработки дескрипторы файлов должны передаваться как typeglobs или ссылки на typeglobs: строки считаются именами файлов.

При использовании нелитеральных значений режима может понадобиться преобразовать восьмеричную строку в число с помощью функции oct. Дело в том, что Perl не считает строку восьмеричным числом лишь потому, что она начинается символом «0».

```
$DEF_MODE = 0644; # Здесь нельзя использовать кавычки!
PROMPT: {
    print "Новый режим? ";
    $strmode = <STDIN>;
    exit unless defined $strmode; # проверка eof
    if ($strmode =~ /^\s*$/) {    # проверка пустой строки
        $mode = $DEF_MODE;
    }
    elsif ($strmode !~ /^\d+$/) {
        say "Требуется числовой режим, а не $strmode";
        redo PROMPT;
    }
    else {
        $mode = oct($strmode);    # преобразует "755" в 0755
    }
    chmod $mode, @files;
}
```

Эта функция работает с числовыми режимами во многом подобно системному вызову UNIX *chmod*(2). Чтобы воспользоваться символьным интерфейсом вроде того, что предоставляет команда *chmod*(1), обратитесь к модулю File::chmod в CPAN.

Можно также импортировать константы S_I* из модуля Fcntl:

```
use Fcntl ":mode";
chmod S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH, @executables;
```

Некоторые считают, что это читается легче, чем 0755. Ну, что тут скажешь.

chomp




```
chomp VARIABLE
chomp LIST
chomp
```

(Как правило) удаляет замыкающий символ перевода строки из значения строковой переменной. Это несколько более безопасная версия chop (ее мы опишем ниже), поскольку не изменяет строку, которая не оканчивается символом перевода строки. Точнее, она удаляет строку-терминатор соответственно текущему значению $/, а не просто любой последний символ.

В отличие от chop, функция chomp возвращает число удаленных символов. Если значением $/ является "" (режим абзацев), chomp удаляет все замыкающие символы перевода строки из указанной строки (или строк, если аргументом является *LIST*). В режиме поглощения ($/ = undef) или в режиме обработки записей фикси-

рованной длины ($/ – ссылка на целое число), chomp не делает ничего. Нельзя применять chomp к литералам – только к переменным. Если применить ее к хешам, будут удалены значения, но не ключи.

Например:

```
while (<PASSWD>) {
    chomp; # игнорировать \n в последнем поле
    @array = split /:/;

}
```

Фильтрам ввода/вывода разрешено переопределять значение переменной $/ и размечать точки усечения строк. Преимущество в том, что фильтры ввода/вывода способны распознавать несколько видов окончаний строк (например, разделители абзацев и строк Юникода), но, тем не менее, chomp обеспечивает безопасное удаление окончания текущей строки.

Функция chomp в настоящее время недостаточно сообразительна, чтобы обрабатывать последовательности разрыва строк Юникода, соответствующие метасимволу \R в регулярных выражениях. Обеспечить их поддержку можно таким способом:

```
s/\R/\n/g, # преобразовать все разрывы строк Юникода в \n
```

Или иногда таким:

```
my @paras = split /\R+/, our $file_contents,
```

Однако если необходимо сохранить последовательности разрывов строк, лучше использовать такой прием:

```
our $line =~ s/(\R?)\z//,
my  $terminator = $1;
```

chop




```
chop VARIABLE
chop LIST
chop
```

Эта функция «обрубает» последний символ строковой переменной и возвращает его. Оператор chop применяется в основном для удаления символа перевода строки в конце прочитанной записи, и работает быстрее, чем операция подстановки. Если это все, что вам нужно, то безопаснее применять chomp, поскольку chop всегда укорачивает строку, независимо от того, что в ней находилось; кроме того, chomp более разборчива.

Нельзя применять chop к литералам – только к переменным. Если chop применяется к списку переменных, усекается каждая строка в списке:

```
@lines = `cat myfile`;
chop @lines;
```

chop можно применить к любому левостороннему значению (lvalue), в том числе к операции присваивания:

```
chop($cwd = `pwd`);
chop($answer = <STDIN>);
```

Это не то же, что

```
$answer = chop($tmp = <STDIN>); # НЕВЕРНО
```

где в $answer сохраняется перевод строки, потому что chop возвращает «отрубленный» символ, а не оставшуюся строку (которая находится в $tmp). Один из способов получить предполагавшийся результат состоит в применении substr:

```
$answer = substr <STDIN>, 0, -1;
```

Но чаще это записывается так:

```
chop($answer = <STDIN>);
```

В самом общем случае chop можно выразить через substr:

```
$last_char = chop($var);
$last_char = substr($var, -1, 1, ""); # то же самое
```

Поняв эту эквивалентность, можно удалять больше одного символа. Для этого следует использовать substr в качестве левостороннего значения, присваивая ему пустую строку. Следующий код удаляет последние пять символов в $caravan:

```
substr($caravan, -5) = "";
```

Отрицательный индекс заставляет substr вести отсчет от конца строки, а не от начала. Сохранить удаленные таким способом символы можно при помощи варианта substr с четырьмя аргументами, создав некий пятикратный вариант chop:

```
$tail = substr($caravan, -5, 5, "");
```

Все это достаточно ненадежно, ведь операции выполняются над кодами, а не графемами. В Perl нет настоящего режима работы с графемами, поэтому вам придется решать эту проблему самостоятельно. Возьмем в качестве примера слово *naïveté*, которое в NFD имеет вид nai\x{308}vete\x{301}. Если применить к нему функцию chop, вы получите не *naïvet*, а *naïvete*. Для отсечения графем, а не кодов, следует использовать s/\X\z//. Значительную помощь во всем этом может оказать модуль Unicode::GCString из CPAN.

chown




```
chown LIST
```

Изменяет владельца и группу для списка файлов. Первые два элемента списка должны быть *числовыми* UID и GID, причем именно в этом порядке. Значение -1 в любой позиции интерпретируется большинством систем как указание оставить элемент без изменений. Функция возвращает число успешно модифицированных файлов. Например:

```
chown($uidnum, $gidnum, "file1", "file2") == 2
        || die "невозможно chown file1 или file2: $!";
```

установит $cnt в 0, 1 или 2, в зависимости от того, сколько файлов было изменено (в смысле успеха операции, а не в смысле того, что изменился владелец). Вот более типичное применение:

```
chown($uidnum, $gidnum, @filenames) == @filenames
        || die "невозможно выполнить chown для @filenames. $!";
```

Следующая подпрограмма принимает имя пользователя, выясняет идентификаторы пользователя и группы, а затем выполняет chown:

```
sub chown_by_name {
    my($user, @files) = @_;
    chown((getpwnam($user))[2,3], @files) == @files
          || die "невозможно выполнить chown для @files: $!"
}

chown_by_name("fred", glob("*.c"));
```

Однако иногда бывает нежелательно изменять группу, как это делает предыдущая функция, потому что в файле */etc/passwd* каждый пользователь ассоциируется с единственной группой, даже при том, что пользователь может быть членом многих вторичных групп в соответствии с */etc/group*. Выход в том, чтобы передать -1 в качестве GID, в результате чего группа не изменится. Если передать -1 в качестве UID, а также допустимый GID, можно установить группу, не меняя владельца.

В системах, поддерживающих *fchown*(2), в списке аргументов можно также передавать дескрипторы файлов. В системах, не поддерживающих *fchown*(2), передача дескрипторов файлов вызовет исключение во время выполнения. Для корректной обработки, дескрипторы файлов должны передаваться как typeglobs или ссылки на typeglobs: строки считаются именами файлов.

В большинстве систем менять владельца файла разрешается только суперпользователю, однако простой пользователь должен быть в состоянии изменить группу на любую из своих вторичных групп. В недостаточно защищенных системах эти ограничения могут быть ослаблены, но такое допущение ведет к созданию непереносимого кода. В POSIX-системах можно определить, какое правило используется, следующим образом:

```
use POSIX qw(sysconf _PC_CHOWN_RESTRICTED);
# попытаться выполнить, если сценарий выполняется от имени
# суперпользователя или в системе с ослабленными ограничениями
if ($> == 0 || !sysconf(_PC_CHOWN_RESTRICTED) ) {
    chown($uidnum, -1, $filename)
        || die "невозможно chown $filename на $uidnum: $!
}
```

chr

$_

```
chr NUMBER
chr
```

Возвращает символ, представленный в наборе символов числом *NUMBER* (усеченным до целого). Например, вызов chr(65) вернет символ «A», LATIN SMALL LETTER A, и в ASCII, и в Юникоде, а chr(0x2122) вернет символ «™», TRADE MARK SIGN. Обратную операцию выполняет функция ord.

Если *NUMBER* – отрицательное число, функция вернет символ Юникода REPLACEMENT CHARACTER, U+FFFD.[1]

---

[1] Только в отсутствие прагмы bytes, в области действия которой используются только младшие восемь битов значения.

(Имейте в виду, что символы с кодами в диапазоне от 128 до 255 внутренне не кодируются в UTF-8 из соображений обратной совместимости. Вы это вряд ли заметите, но если все-таки заметите, то будете знать причину.)

Если вы предпочитаете названия символов числам (например, "\N{WHITE SMILING FACE}" для обозначения смайлика Юникода «☺»), загляните в раздел «charnames» главы 29. Преобразовать код символа в его официальное название можно посредством функции charnames::viacode.

chroot




```
chroot FILENAME
chroot
```

При успешном выполнении функции *FILENAME* становится новым корневым каталогом для текущего процесса – отправной точкой маршрутов, начинающихся косой чертой «/». Этот каталог наследуется вызовами exec и всеми подпроцессами, ответвляемыми fork после вызова chroot. Способа отменить chroot нет. По соображениям безопасности эту функцию может вызывать только суперпользователь. Следующий код делает примерно то же, что и многие серверы FTP:

```
chroot((getpwnam("ftp"))[7])
    || die "Не разрешен анонимный ftp: $!"
```

Эта функция, скорее всего, будет работать только в системах UNIX. См. *chroot*(2).

close




```
close FILEHANDLE
close
```

Закрывает файл, сокет или канал, связанный с дескриптором *FILEHANDLE*, предварительно вытолкнув все буферы ввода/вывода. Если аргумент опущен, закрывает текущий выбранный дескриптор. Возвращает истинное значение, если дескриптор файла был закрыт успешно, и ложное в противном случае. Не требуется закрывать *FILEHANDLE*, если сразу за этим предполагается открыть его – очередной вызов open закроет дескриптор автоматически. (См. open.) Однако явное применение close для файла ввода влечет сброс счетчика строк ($.), тогда как неявное закрытие с помощью open – нет.

FILEHANDLE может быть выражением, значение которого способно послужить косвенным дескриптором файла (это будет реальное название дескриптора файла, либо ссылка на то, что можно интерпретировать как объект дескриптора файла).

Если дескриптор файла получен в результате открытия канала, вызов close возвращает ложное значение при неудаче какого-либо из выполняющихся в нем системных вызовов или если программа на другом конце канала завершила работу с ненулевым кодом. В последнем случае close устанавливает $! ($OS_ERROR) в нуль. Поэтому, если close для канала возвращает ненулевое значение, проверьте $!, чтобы выяснить, была ли проблема связана с самим каналом (ненулевое значение) или с программой на другом конце (нулевое значение). В любом случае $? ($CHILD_ERROR) содержит значение кода ожидания (см. его интерпретацию в описании функции system) команды, связанной с другим концом канала. Например:

```
open(OUTPUT, "| sort -rn | lpr -p") # канал для sort и lpr
    || die "Невозможно запустить канал sortlpr: $!";
print OUTPUT @lines;                # вывести что-то в output
close OUTPUT                        # ждать завершения sort
    || warn $! ? "Системная ошибка при закрытии канала sortlpr  $!"
              : "Код ожидания $? от канала sortlpr";
```

Дескриптор файла, полученный в результате вызова *dup*(2) для канала, считается обычным дескриптором файла, поэтому close не будет ждать завершения порожденного процесса по этому дескриптору. Ожидание следует реализовывать посредством закрытия исходного дескриптора файла. Например:

```
open(NETSTAT, "netstat -rn |")
    || die "невозможно выполнить netstat: $!";
open(STDIN, "<&NETSTAT")
    || die "невозможно выполнить dup в stdin: $!"
```

Если закрыть здесь STDIN, то ожидания нет, а если NETSTAT, то есть.

Если каким-то образом ухитриться самостоятельно удалить завершившийся порожденный процесс, попытка закрыть канал, связывающий с ним, окажется неудачной. Это может произойти, если у вас есть собственный обработчик $SIG{CHLD}, который запускается при завершении порожденного процесса канала, или вы намеренно вызываете waitpid с идентификатором процесса, полученным при вызове open.

closedir




```
closedir DIRHANDLE
```

Закрывает каталог, открытый с помощью opendir, и возвращает признак успеха операции. См. примеры в описании readdir. Параметр *DIRHANDLE* может быть выражением, значение которого может использоваться как косвенный дескриптор каталога, обычно являющийся настоящим именем дескриптора каталога.

connect




```
connect SOCKET, NAME
```

Инициирует соединение с другим процессом, который находится в ожидании в вызове accept. Возвращает истинное значение в случае успеха и ложное в противном случае. Аргумент *NAME* должен быть упакованным сетевым адресом сокета надлежащего типа. Приведем пример, предполагающий, что *SOCKET* является ранее созданным сокетом:

```
use Socket;

my ($remote, $port) = ("www.perl.com", 80);
my $destaddr = sockaddr_in($port, inet_aton($remote));
connect SOCK, $destaddr
    || die "Can't connect to $remote at port $port: $!";
```

Чтобы отсоединить сокет, используйте close или shutdown. См. также примеры в разделе «Сокеты» главы 15. См. также *connect*(2). В большинстве случаев для сокетов лучше использовать высокоуровневый интерфейс, предоставляемый модулем IO::Socket.

continue

Это – обычная инструкция управления потоком выполнения, а не функция. Если инструкция continue применяется к блоку *BLOCK* (например, в цикле while или foreach), она всегда выполняется непосредственно перед вычислением условного выражения, а в циклах for(;;) она выполняется перед выполнением третьей части инструкции. То есть, ее можно использовать для приращения переменной цикла, даже когда цикл был продолжен инструкцией next (которая напоминает инструкцию continue в языке С).

Внутри блока continue допускается использовать инструкции last, next или redo; last и redo в этом случае будут вести себя, как если бы они были выполнены в основном блоке. То же верно и для инструкции next, но, поскольку она вызовет выполнение блока continue, может сложиться интересная ситуация.

```
while (EXPR) {
    ### redo всегда передает управление сюда
    do_something;
} continue {
    ### next всегда передает управление сюда
    do_something_else;
    # затем переход в начало цикла к проверке EXPR
}
### last всегда передает управление сюда
```

Отсутствие блока continue эквивалентно использованию пустого блока, что достаточно логично, поэтому next передаст управление выражению проверки условия в начале цикла. См. раздел «Операторы циклов» в главе 4.

Однако если включена особенность "switch", continue также является оператором, осуществляющим выход из текущего предложения when или блока default, и, по умолчанию, передает управление следующему предложению. См. раздел «Инструкция given» в главе 4.

cos




```
cos EXPR
cos
```

Возвращает косинус *EXPR* (выраженный в радианах). Например, следующий сценарий выведет таблицу косинусов углов, измеренных в градусах:

```
# "Ленивый" способ перевода градусов в радианы.

$pi = atan2(1,1) * 4;
$pi_over_180 = $pi/180;

# Вывод таблицы.
for ($deg = 0; $deg <= 90; $deg++) {
    printf "%3d %7.5f\n", $deg, cos($deg * $pi_over_180);
}
```

Операция, обратная взятию косинуса, выполняется посредством функции acos() из модуля Math::Trig или POSIX, или при помощи соотношения:

```
sub acos { atan2( sqrt(1 - $_[0] * $_[0]), $_[0] ) }
```

crypt



crypt *PLAINTEXT*, *SALT*

Вычисляет необратимый хэш-код строки точно в стиле *crypt*(3). Это до некоторой степени полезно для проверки файла паролей на наличие нестойких паролей,[1] хотя на самом деле задача в первую очередь заключается в том, чтобы не допустить добавления плохих паролей.

Функция crypt необратима по замыслу, что можно сравнить с разбиванием яиц для приготовления омлета. Не существует (известного) способа расшифровать зашифрованный пароль, кроме исчерпывающего перебора.

При проверке зашифрованной строки передавайте зашифрованный текст в аргументе *SALT* (например, crypt($plain, $crypted) eq $crypted). Это позволит программе работать как со стандартной версией crypt, так и с более экзотическими реализациями.

Выбирая новое значение *SALT*, необходимо, как минимум, создать случайную строку из двух символов, принадлежащих множеству [./0-9A-Za-z] (например, путем join "", (".","/", 0..9, "A".."Z", "a".."z")[rand 64, rand 64]). В старых реализациях crypt требовались только первые два символа из *SALT*, но код, дающий только два первые символа, считается теперь непереносимым. Интересные подробности можно найти на странице руководства для *crypt*(3).

Следующий пример проверяет – знает ли тот, кто работает с этой программой, свой пароль:

```
$pwd = (getpwuid ($<))[1]; # Предполагается, что мы в UNIX.

system "stty -echo";       # или см. Term::ReadKey в CPAN
print "Пароль: ";
chomp($word = <STDIN>);
print "\n";
system "stty echo";

if (crypt($word, $pwd) ne $pwd) {
    die "Извини...\n";
} else {
    print "Порядок\n";
}
```

Разумеется, вводить свой пароль по просьбе всякого, кто об этом попросит, неблагоразумно.

Теневые файлы паролей несколько более безопасны, чем традиционные файлы паролей, и для доступа к ним могут потребоваться права суперпользователя. Поскольку мало какие программы должны выполняться с такими широкими полномочиями, можно сделать так, чтобы программа поддерживала собственную независимую систему идентификации, сохраняя обработанные crypt строки в файлах, отличных от */etc/passwd* и */etc/shadow*.

Функция crypt не пригодна для шифрования больших объемов данных хотя бы потому, что вы не сможете получить свою информацию обратно. Загляните в ка-

---

[1] Это разрешается делать только тем, кто имеет достойные намерения.

талоги Crypt::*, Digest::* и PGP::* на своем любимом зеркале CPAN, и вы обнаружите множество модулей, которые могут оказаться полезными.

При шифровании строк Юникода, которые могут содержать символы с кодами выше 255, Perl попытается скопировать эту строку в строку 8-разрядных байтов и только потом вызовет crypt, передав ей копию. Если этот прием сработает, хорошо. А если нет, crypt возбудит исключение.

dbmclose

```
dbmclose HASH
```

Разрывает связь между файлом DBM (базы данных) и хешем.

dbmclose в действительности является просто вызовом untie с надлежащими аргументами, но оставлена для совместимости со старыми версиями Perl.

dbmopen



```
dbmopen HASH, DBNAME, MODE
```

Связывает файл DBM с хешем (т.е. ассоциативным массивом). (DBM означает «database management» – управление базой данных – и состоит из набора библиотечных подпрограмм на языке C, обеспечивающих произвольный доступ к записям с использованием алгоритма хеширования.) *HASH* является именем хеша (вместе с %). *DBNAME* – имя базы данных (без каких-либо расширений, таких как *.dir* или *.pag*). Если база данных не существует и задан допустимый режим *MODE*, dbmopen создаст файл базы данных с правами доступа, определяемыми аргументом *MODE* и значением маски umask. Чтобы предотвратить создание новой базы данных, в аргументе *MODE* можно передать значение undef, и функция вернет ложное значение, если не сможет найти указанную базу данных. Значения, присвоенные хешу до вызова dbmopen, окажутся недоступными.

Функция dbmopen в действительности просто вызывает tie с надлежащими аргументами и оставлена для совместимости с «древними» версиями Perl. Функция dbmopen возвращает значение, полученное от функции tie: связанный объект в случае успеха и ложное значение в случае неудачи. Управлять выбором используемой библиотеки DBM можно непосредственно с помощью интерфейса tie, либо путем загрузки соответствующего модуля перед вызовом dbmopen. Вот пример, который работает на некоторых системах с версиями DB_File, сходными с версиями в браузере Netscape:

```
use DB_File;
dbmopen(%NS_Hist, "$ENV{HOME}/.netscape/history.dat", undef)
    || die "Невозможно открыть файл истории netscape: $!"

while (($url, $when) = each %NS_Hist) {
    next unless defined($when);
    chop ($url, $when);        # удалить замыкающие нулевые байты
    printf "Visited %s at %s.\n", $url,
        scalar(localtime(unpack("V".$when)));
}
```

Если у пользователя нет права записи в файл DBM, он сможет только читать из хеша. Выяснить наличие права на запись можно посредством проверки вида -w $file,

а можно также попытаться выполнить запись в хеш внутри инструкции eval {}, которая перехватит исключение.

Такие функции, как keys и values, могут возвращать очень большие списки значений при работе с большими файлами DBM. Более предпочтительным для обхода записей в больших файлах DBM может оказаться применение функции each, позволяющей не загружать в память файл целиком.

Для хешей, связанных с файлами DBM, действуют ограничения, накладываемые используемым пакетом DBM, в том числе ограничение объема размещаемых данных. В случае работы с короткими ключами и значениями такие проблемы возникают редко. См. также описание модуля DB_File.

Следует также помнить, что многие существующие базы данных DBM содержат ключи и значения, заканчивающиеся пустым символом, поскольку создавались для программ на C. Например, файл журнала Netscape и старый файл псевдонимов *sendmail*. В таких случаях следует использовать "$key\0" для извлечения значения, а затем удалять из него пустой символ:

```
$alias = $aliases{"postmaster\0"};
$alias =~ s/\0\z//;          # удалить пустой символ
```

Начиная с версии v5.8.4 строки, завершающиеся пустыми символами, можно обрабатывать автоматически, с помощью стандартного модуля DBM_Filter.

```
use DB_File;
$db = dbmopen(%aliases, "/etc/mail/aliases", undef)
    || die "невозможно dbmopen /etc/mail/aliases: $!";
$db->Filter_Push("null");
$alias = $aliases{"postmaster"};
print "postmaster является псевдонимом $alias\n";
```

Аналогичную стратегию с успехом можно использовать при добавлении фильтра utf8 в дескриптор файла. Примеры использования строк Юникода в качестве ключей и значений в файлах DBM можно найти в главе 6.

В настоящее время нет универсального встроенного средства блокировки файлов DBM. Некоторые считают это ошибкой. Модуль GDBM_File делает попытку обеспечить блокировку на уровне файла в целом. Если есть сомнения, лучше использовать отдельный файл блокировки.

defined




```
defined EXPR
defined
```

Возвращает логическое значение, говорящее о том, имеет *EXPR* определенное значение или нет. По большей части данные, с которыми вы имеете дело, являются определенными, но скаляр, который не содержит допустимого строкового, числового или ссылочного значения, считается содержащим неопределенное значение, или, для краткости, undef. Инициализация скалярной переменной конкретным значением определяет ее, и она остается определенной, пока не получит неопределенное значение посредством присваивания или вызова undef.

Многие операции возвращают undef в случае исключения, например, когда достигнут конец файла, использовано значение неопределенной переменной, возникла

ошибка операционной системы и т.д. Поскольку undef – это разновидность ложного значения, простая логическая проверка не различает undef, числовой ноль, пустую строку и строку из одного символа "0" – все они являются одинаково ложными. Функция defined позволяет отличить неопределенную пустую строку от определенной пустой строки, когда применяются операторы, которые действительно могут возвращать пустую строку.

Следующий фрагмент проверяет скалярное значение из хеша:

```
print if defined $switch{D};
```

При таком использовании с элементом хеша defined только сообщает, является ли значение определенным, но не свидетельствует о наличии или отсутствии ключа в хеше. Допустимо иметь ключ, значение которого является неопределенным, и при этом сам ключ существует. Используйте exists, чтобы определить, существует ли ключ.

В следующем примере действует соглашение, в соответствии с которым некоторые операции возвращают неопределенное значение, когда кончаются данные (предполагается, что значение undef не является допустимым):

```
print "$val\n" while defined($val = pop(@ary));
```

А в следующем примере мы делаем то же самое с функцией getpwent, чтобы извлечь информацию о пользователях системы.

```
setpwent();
while (defined($name = getpwent())) {
    say "<<$name>>";
}
endpwent();
```

Тот же прием можно использовать для проверки успешности системных вызовов, которые могут возвращать ложное значение:

```
die "Невозможно readlink $sym: $!"
    unless defined($value = readlink $sym);
```

С помощью defined можно также проверить, определена ли подпрограмма. Это позволит избежать аварии при вызове несуществующих подпрограмм (или подпрограмм, которые объявлены, но не имеют определения):

```
indir("funcname", @arglist);
sub indir {
    my $subname = shift;
    no strict "refs";     # чтобы использовать subname косвенно
    if (defined &$subname) {
        &$subname(@_);    # или $subname->(@_);
    }
    else {
        warn "Игнорирую вызов недопустимой функции $subname";
    }
}
```

Однако даже неопределенные подпрограммы могут быть доступны для вызова, если в пакете имеется функция AUTOLOAD, обслуживающая вызовы неопределенных функций в этом пакете.

Не рекомендуется использовать defined с объектами составными (хешами и массивами), поскольку для них defined сообщает лишь о выделении памяти под объект. Вместо этого лучше просто проверять число элементов массива или хеша:

```
if (@an_array) { print "массив содержит элементы\n" }
if (%a_hash)   { print "хеш содержит элементы\n" }
```

Будучи вызванной для элемента хеша, defined сообщает, является ли значение определенным, но не свидетельствует о наличии или отсутствии ключа в хеше. Для проверки наличия ключа используйте exists.

См. также undef и exists.

delete

```
delete EXPR
```

Удаляет элемент (или срез элементов) из указанного хеша или массива. (См. unlink, если требуется удалить файл.) Удаленные элементы возвращаются в указанном порядке, хотя для связанных переменных, например файлов DBM, это не гарантируется. После операции удаления функция exists возвращает ложное значение для любого удаленного ключа или индекса. (Для сравнения: после вызова undef функция exists продолжает возвращать истинное значение, поскольку функция undef лишь присваивает элементу неопределенное значение, но не удаляет его.)

Удаление из хеша %ENV модифицирует среду выполнения. Удаление из хеша, связанного с файлом DBM (доступным для записи), удаляет запись из этого файла DBM.

Удаление из массива ведет к тому, что элемент в указанной позиции возвращается в полностью неинициализированное состояние, но образующийся разрыв при этом не закрывается, так как это изменило бы позиции последующих элементов. Чтобы получить массив без разрывов, используйте функцию splice. Однако если удалить последний элемент массива, длина массива сократится на единицу или более, в зависимости от положения нового последнего элемента, если он есть.

Вызывать delete для массивов не рекомендуется, и такая возможность, вероятно, будет устранена в будущем.

EXPR может иметь произвольную сложность при условии, что последняя операция является поиском в хеше или массиве:

```
# создать массив массивов из хеша
$dungeon[$x][$y] = \%properties;

# удалить одно свойство из хеша
delete $dungeon[$x][$y]{"OCCUPIED"}

# удалить сразу три свойства из хеша
delete @{ $dungeon[$x][$y] }{ "OCCUPIED", "DAMP", "LIGHTED" };

# удалить ссылку на %properties из массива
delete $dungeon[$x][$y];
```

В следующем примитивном примере происходит неэффективное удаление всех значений из %hash:

```
for my $key (keys %hash) {
    delete $hash{$key};
}
```

Так же как и в этом:

```
delete @hash{keys %hash};
```

В обоих случаях результат достигается медленнее, чем в случае присваивания пустого списка (если сделать хеш неопределенным, тоже получится быстрее):

```
%hash = ();  # полностью очистить %hash
undef %hash; # забыть, что %hash когда-либо существовал
```

Аналогично, для массивов:

```
for my $index (0 .. $#array) {
    delete $array[$index];
}
```

и:

```
delete @array[0 .. $#array];
```

менее эффективны, чем любая из этих операций:

```
@array = ();  # полностью очистить @array
undef @array; # забыть, что @array когда-либо существовал
```

Для локализованного удаления элементов массива или хеша в текущем блоке можно также использовать конструкцию delete local *EXPR*. Локально удаленные элементы прекращают свое существование только до выхода из блока.

die




```
die LIST
die
```

Вне eval эта функция выводит значение, составленное из элементов из списка *LIST*, в STDERR и завершает программу с текущим значением $! (переменная errno C-библиотеки). Если $! равна 0, выход производится со значением $? >> 8 (что является значением кода завершения последнего порожденного процесса, полученного из system, wait, close для канала или \`command\`). Если значение $? >> 8 равно 0, осуществляется выход со значением 255.

Внутри eval эта функция записывает в переменную $@ сообщение об ошибке и выходит из eval, возвращая undef. Функция die может, таким образом, применяться для создания именованных исключений, обрабатываемых на более высоком уровне программы. См. описание функции eval далее в этой главе.

Если *LIST* – единственная ссылка на объект, предполагается, что это объект исключения, и он возвращается в исходном виде как исключение в $@ (описывается ниже).

Если список *LIST* пуст, а $@ уже содержит строковое значение (например, оставшееся от предшествующего вызова eval), то это значение повторно используется после добавления "\t...propagated". Это полезно для распространения (повторного возбуждения) исключений:

```
eval { ... };
die unless $@ =~ /Ожидаемое исключение/;
```

Если список *LIST* пуст, а $@ уже содержит объект исключения, вызывается метод $@->PROPAGATE с дополнительными параметрами, именем файла и строкой в файле, чтобы определить, как должна распространяться ошибка, и возвращаемое значение сохраняется в переменную $@. То есть как если бы была выполнена инструкция if $@ = eval { $@->PROPAGATE(__FILE__, __LINE__) }.

Если LIST пуст и $@ пуста, используется строка "Died". Если необработанное исключение привело к завершению интерпретатора, код завершения определяется из значений $! и $? с помощью следующего псевдокода:

```
exit $! if $!;              # errno
exit $? >> 8 if $? >> 8;    # код завершения дочернего процесса
exit 255;                   # крайний случай
```

Цель состоит в том, чтобы уместить как можно больше информации о вероятной причине в ограниченное пространство системного кода завершения. Однако, поскольку $! может быть установлена любым системным вызовом, значение кода завершения, использованного функцией die, может оказаться непредсказуемым, поэтому не следует заниматься его интерпретацией; лучше ограничиться фактом отличия этого значения от нуля.

Если последнее значение в *LIST* не оканчивается символом перевода строки (и вы не передаете объект исключения), к сообщению дописываются имя файла текущего сценария, номер строки и номер вводимой строки (если такая есть), а также символ перевода строки. Совет: иногда можно сделать сообщение более осмысленным, добавив в него ", stopped", если далее в сообщение будет добавлена такая строка, как "at scriptname line 123". Допустим, что выполняется сценарий *canasta*; посмотрите на разницу между двумя способами использования die:

```
die "/usr/games is no good";
die "/usr/games is no good, stopped";
```

порождающими, соответственно:

```
/usr/games is no good at canasta line 123
/usr/games is no good, stopped at canasta line 123.
```

Чтобы использовать собственные сообщения об ошибках с именами файлов и номерами строк, используйте специальные лексемы __FILE__ и __LINE__ (которые не интерполируются в строки):

```
die sprintf qq("%s" строка "%s", тьфу на тебя!\n),
    __FILE__, __LINE__;
```

В результате получится такой вывод:

```
"canasta", строка 38, тьфу на тебя!
```

Еще одно замечание по поводу стиля: посмотрите на следующие эквивалентные примеры:

```
die "Can't cd to spool: $!" unless chdir "/usr/spool/news";

chdir("/usr/spool/news")        || die "Can't cd to spool: $!"
```

Поскольку важной частью является chdir, вторая форма в целом предпочтительнее.

Функцию die можно также вызвать со ссылкой, а если заключить этот вызов в eval, то $@ будет содержать эту ссылку. Это позволяет вовлекать в обработку исключений объекты, способные хранить любую информацию об исключении. Такая схема иногда предпочтительнее сопоставления строковых значений из $@ с регулярными выражениями. Поскольку переменная $@ является глобальной, а eval может использоваться в реализации объекта, будьте внимательны, чтобы при анализе не затереть ссылку на объект ошибки в глобальной переменной. Это легко обеспечивается созданием локальной копии перед началом манипуляций. Например:

```
use Scalar::Util "blessed";

eval { WHATEVER; die Some::Module::Exception->new( FOO => "bar" ) };
if (my $eval_err = $@) {
    if (blessed($eval_err) && $eval_err->isa("Some::Module::Exception")) {
        # обработать Some::Module::Exception
    }
    else {
        # обработать любые другие исключения
    }
}
```

Поскольку перед выводом все неперехваченные исключения превращаются интерпретатором в строки, у вас, возможно, появится желание переопределить в объекте-исключении операцию преобразования в строку. Подробности – в главе 13.

Можно определить функцию, которая будет вызываться непосредственно перед die, сохранив такую функцию в $SIG{__DIE__}. Данной функции-обработчику будет передаваться текст сообщения об ошибке, который может быть изменен обработчиком (если необходимо) повторным вызовом die. Лишь опытные и отчаянные маги предпринимают попытку сотворить такое волшебство, и мало кто из них остается в живых.

См. также описания eval, exit, warn, %SIG, прагмы warnings и модуля Carp.

do (block)

```
do BLOCK
```

Форма do *BLOCK* выполняет последовательность команд в блоке и возвращает значение последнего выражения, вычисленного в блоке. Если присутствует модификатор while или until, Perl выполнит *BLOCK* один раз, прежде чем проверить условие цикла. (В других командах модификаторы цикла сначала проверяют условие.) Сама конструкция do *BLOCK* циклом *не* считается, поэтому команды управления циклом next, last и redo нельзя применять для выхода из блока или его перезапуска. О том, как обойти это ограничение, читайте в разделе «Голые блоки» главы 4.

do (file)

$! T X̲T̄

```
do FILE
```

Форма do *FILE* использует значение *FILE* как имя файла и выполняет содержимое файла как сценарий Perl. Ее основным назначением является (или, скорее, являлось) включение подпрограмм из библиотеки, поэтому:

```
do "stat.pl";
```

имеет большое сходство с:

```
scalar eval `cat stat.pl`; # `type stat.pl` в Windows
```

за исключением того, что do более эффективна, более выразительна, использует имя текущего файла для сообщений об ошибках, выполняет поиск в каталогах, перечисленных в массиве @INC, и обновляет %INC, если файл найден. (См. главу 25.) Разница еще и в том, что код, выполняемый с помощью do *FILE*, не видит лексические переменные из охватывающей лексической области видимости, тогда как код в eval *FILE* видит их. Они одинаковы в том, что заново производят анализ файла при каждом вызове; поэтому нежелательно делать это в цикле, если только само имя файла не меняется при каждом проходе цикла.

Если do не может прочесть файл, возвращается undef, и $! присваивается ошибка. Если do может прочесть файл, но не может его скомпилировать, возвращается undef, а сообщение об ошибке записывается в $@. Если файл успешно скомпилирован, do возвращает значение последнего вычисленного выражения.

Включение библиотечных модулей (имеющих обязательный суффикс *.pm*) лучше выполнять с помощью операторов use и require, которые также проверяют ошибки и возбуждают исключение при возникновении ошибки. Они обеспечивают и другие преимущества: позволяют избежать повторной загрузки, содействуют объектно-ориентированному программированию и дают советы компилятору по прототипам функций.

Но do *FILE* все еще может пригодиться, например, для чтения файлов конфигурации программы. Ручную проверку ошибок можно произвести так:

```
# прочитать файлы конфигурации, сначала системные, затем пользовательские
for $file ("/usr/share/proggie/defaults.rc",
           "$ENV{HOME}/.someprogrc")
{
    unless ($return = do $file) {
        warn "couldn't parse $file: $@" if $@;
        warn "couldn't do $file: $!"    unless defined $return;
        warn "couldn't run $file"       unless $return;
    }
}
```

Долго работающий демон может периодически проверять временную метку на своем файле конфигурации, и, если файл изменился после предыдущей попытки чтения, вызывать do, чтобы перезагрузить этот файл. do позволяет реализовать это более аккуратно, чем require или use.

do (subroutine)




```
do SUBROUTINE(LIST)
```

do *SUBROUTINE*(*LIST*) – устаревшая форма вызова подпрограммы. Если *SUBROUTINE* не определена, возбуждает исключение. См. главу 7.

dump

```
dump LABEL
dump
```

Вызывает немедленный вывод дампа памяти. В основном это нужно, чтобы с помощью программы *undump* (не поставляется) превратить дамп памяти в исполняемый двоичный файл после инициализации своих переменных в начале программы. При запуске такого двоичного файла сначала выполняется goto *LABEL* (со всеми ограничениями, присущими goto). Считайте, что dump *LABEL* – это инструкция перехода goto, в ходе которой выполняется дамп памяти и происходит перевоплощение. Если *LABEL* опущена, программа перезапускается с начала. Внимание: все файлы, открытые в момент создания дампа, не будут открыты при перевоплощении программы, что может запутать Perl. См. также описание ключа командной строки *-u* в главе 17.

Эта функция сейчас значительно устарела, потому что крайне трудно в общем случае преобразовать дамп памяти в исполняемый файл, а также потому, что ее заменили различные серверы компиляции для генерации переносимого байт-кода и компилируемого C-кода. Однако люди, управляющие проектом развития компилятора Perl (*perlcc* сотоварищи) в CPAN, сообщают, что поддержка dump и *undump* вскоре может воскреснуть.

Если вы планирует использовать dump для повышения скорости выполнения программы, изучите вопросы эффективности, рассмотренные в главе 21, а также генератор кода Perl в главе 16. Можно также рассмотреть возможность автоматической загрузки и самозагрузки, благодаря которым, по крайней мере, покажется, что программа выполняется быстрее.

each




```
each HASH
each ARRAY
each EXPR
```

Обходит элементы хеша, выбирая на каждом шаге одну пару ключ/значение. В списочном контексте each возвращает двухэлементный список, состоящий из ключа и значения для следующего элемента хеша, что позволяет обойти хеш. В скалярном контексте each возвращает просто ключ очередного элемента в хеше. Когда хеш полностью прочитан, возвращается пустой список, который при присваивании создает ложное значение в скалярном контексте, например при проверке условия цикла. После этого следующий вызов each начинает обход хеша сначала. Вот пример типичного применения, в котором используется предопределенный хеш %ENV:

```
while (($key,$value) = each %ENV) {
    print "$key=$value\n";
}
```

Внутри хеша записи расположены в порядке, кажущемся случайным. Функция each соблюдает этот порядок, поскольку каждый хеш помнит, какая запись была прочитана последней. Фактический порядок этой последовательности может измениться в будущих версиях Perl, но гарантируется, что это будет тот же порядок, в каком функция keys (или values) возвращает результаты для того же хеша.

Из соображений безопасности этот порядок может изменяться между прогонами программы.

Для каждого хеша существует единственный итератор, совместно используемый всеми вызовами функций each, keys и values в программе; его можно сбросить, прочтя все элементы хеша или вычислив keys %hash или values %hash. Если во время итераций производится добавление или удаление элементов хеша, его поведение становится не вполне определенным: элементы могут пропускаться или повторяться.

Начиная с версии v5.12, аргументом each может также служить массив. Роль ключей в этом случае играют индексы. В отличие от хеша, для массивов пары будут возвращаться в порядке возрастания ключей (индексов массива).

Начиная с версии v5.14, аргументом each может быть ссылка на «неосвященный» хеш или массив. Такая ссылка разыменовывается автоматически. Эта особенность each считается экспериментальной. Конкретный алгоритм ее работы может измениться в будущем.

```
while (($key,$value) = each $hashref) {
```

См. также keys, values и sort.

eof



```
eof FILEHANDLE
eof()
eof
```

Возвращает истинное значение, если следующая попытка чтения из дескриптора файла *FILEHANDLE* вернет конец файла или если дескриптор *FILEHANDLE* не открыт. *FILEHANDLE* может быть выражением, значение которого определяет действительный дескриптор файла, либо ссылкой на некоторый объект типа дескриптора файла. Функция eof без аргументов сообщает, был ли достигнут конец файла при выполнении последней операции чтения из файла. Функция eof() с пустыми круглыми скобками () проверяет дескриптор ARGV (который чаще всего встречается как пустой дескриптор файла в <>). Поэтому внутри цикла while (<>) функция eof() со скобками обнаружит конец только последнего файла группы. Используйте eof (без скобок) для проверки *каждого* файла в цикле while (<>). Например, следующий код вставляет черточки перед последней строкой последнего файла:

```
while (<>) {
    if (eof()) {
        say "-" x 30;
    }
    print;
}
```

Напротив, следующий сценарий сбрасывает нумерацию строк для каждого прочитанного файла:

```
# сбрасывать нумерацию строк для каждого входного файла
while (<>) {
    next if /^\s*#/;    # пропустить комментарии
    print "$.\t$_";
} continue {
```

```
    close ARGV if eof; # Не eof()!
}
```

Подобно "$" в программе *sed*, функция eof может появляться в диапазонах номеров строк. Следующий фрагмент выведет строки от /pattern/ до конца каждого входного файла:

```
while (<>) {
    print if /pattern/ .. eof;
}
```

Здесь триггерный оператор (..) выполняет поиск по шаблону в каждой строке. Пока шаблон не найден, оператор возвращает ложное значение. Когда, наконец, поиск даст результат, оператор начинает возвращать истинное значение, вызывая печать строк. Когда, наконец, оператор eof вернет истинное значение (в конце исследуемого файла), оператор триггера сбросится и снова начнет возвращать ложное значение для следующего файла в @ARGV.

Предупреждение: функция eof считывает байт и затем возвращает его во входной поток с помощью *ungetc*(3), поэтому в интерактивном контексте от нее нет пользы. На практике опытные программисты Perl редко применяют eof, поскольку различные операторы ввода и так ведут себя прилично в условиях циклов while. См. пример в описании foreach в главе 4.

eval



```
eval BLOCK
eval EXPR
eval
```

Ключевое слово eval служит двум разным, но близким целям, представленным двумя формами синтаксиса: eval *BLOCK* и eval *EXPR*. Первая форма перехватывает исключения (ошибки) этапа выполнения, которые могли бы оказаться фатальными, аналогично блоку «try» в C++ и Java. Вторая форма компилирует и выполняет небольшие фрагменты кода «на лету» на этапе выполнения, а также перехватывает все исключения, как и первая форма. Однако вторая форма выполняется значительно медленнее первой, поскольку каждый раз выполняет синтаксический анализ указанной строки. С другой стороны, она является более универсальной. Какую бы форму вы ни использовали, eval – оптимальный способ любой обработки исключений в Perl.

Любая форма eval возвращает значение последнего вычисленного выражения – в точности, как подпрограммы. Аналогично, чтобы вернуть результат из середины eval, можно использовать оператор return. Выражение, предоставляющее возвращаемое значение, вычисляется в пустом, скалярном или списочном контексте – в зависимости от контекста вызова eval. Дополнительные сведения о том, как определить контекст, можно найти в описании wantarray.

Если возникает перехватываемая ошибка (в том числе полученная в результате вызова функции die), eval возвращает undef и помещает сообщение об ошибке (или объект) в $@. Если ошибки нет, $@ устанавливается равной пустой строке, что обеспечивает надежный способ проверки наличия ошибок. Достаточно простой логической проверки:

```
eval { ..  };    # перехват ошибок этапа выполнения
if ($@) { ... } # обработать ошибку
```

Синтаксис формы `eval` *BLOCK* проверяется на этапе компиляции, чего вполне достаточно. (Если вы знакомы с медленной формой `eval` *EXPR*, вас может смутить это обстоятельство.) Поскольку код в блоке компилируется в то же время, что и код, находящийся за пределами блока, эта форма `eval` не способна перехватывать ошибки синтаксиса.

Форма `eval` *EXPR* может перехватывать синтаксические ошибки, поскольку анализирует код на этапе выполнения. (Если анализ оказывается неудачным, ошибка анализа, как обычно, помещается в `$@`.) В противном случае значение *EXPR* выполняется, как если бы это была маленькая программа на Perl. Код выполняется в контексте текущей программы, что означает доступность для него любых лексических переменных в окружающей области видимости и возможность изменения любых нелокальных переменных – так же, как для подпрограмм и форматов. Код в `eval` интерпретируется как блок, поэтому все переменные с локальной областью видимости, созданные внутри этого блока, действуют только до завершения `eval`. (См. `my` и `local`.) Как и для всякого кода, завершающие точка с запятой в блоке не требуются.

Вот простой командный интерпретатор Perl. Он предлагает пользователю ввести строку с произвольным кодом на Perl, компилирует и выполняет эту строку, и выводит ошибки, если они возникли:

```
print "\nВведите какой-нибудь код на Perl: ";

while (<STDIN>) {
    eval;
    print $@;
    print "\nВведите еше код на Perl: ";
}
```

Вот программа *rename*, осуществляющая групповое переименование файлов:

```
#!/usr/bin/perl
# rename - изменение имен файлов
$op = shift;
for (@ARGV) {
    $was = $_;
    eval $op;
    die if $@;
    # в следующей строке вызывается встроенная функция,
    # а не сценарий с таким же именем
    rename($was,$_) unless $was eq $_;
}
```

Эту программу следует использовать так:

```
% rename 's/\.orig$//'              *.orig
% rename 'y/A-Z/a-z/ unless /^Make/' *
% rename '$_ .= ".bad"'             *.f
```

Поскольку `eval` перехватывает ошибки, которые иначе оказались бы фатальными, с ее помощью удобно определять, реализованы ли некоторые конкретные возможности (например, `fork` или `symlink`).

Поскольку форма `eval BLOCK` подвергается синтаксической проверке на этапе компиляции, обо всех синтаксических ошибках сообщается раньше. Поэтому, если вашим целям одинаково хорошо соответствуют обе формы, `eval EXPR` и `eval BLOCK`, предпочтительнее использовать форму *BLOCK*. Например:

```
# сделать деление на ноль несмертельным
eval { $answer = $a / $b; }; warn $@ if $@;

# то же, но менее эффективно при многократном выполнении
eval '$answer = $a / $b';    warn $@ if $@;

# синтаксическая ошибка этапа компиляции (не перехвачена)
eval { $answer = };          # НЕВЕРНО

# синтаксическая ошибка этапа выполнения
eval '$answer =';            # устанавливает $@
```

Здесь код в *BLOCK* должен быть допустимым кодом на Perl, чтобы пройти фазу компиляции. Код в *EXPR* не интерпретируется до этапа выполнения, поэтому ошибка не возникает, пока не придет время выполнить его.

Блок в `eval BLOCK` не считается циклом, поэтому команды управления циклом `next`, `last` или `redo` нельзя применять для выхода или перезапуска блока.

Конструкция `eval STRING`, выполняемая внутри пакета `DB`, не видит окружающую лексическую область видимости, зато ей доступна область видимости первого фрагмента кода, не имеющего отношения к `DB`. Обычно об этом не приходится беспокоиться, только если вы не собираетесь написать свой отладчик для Perl.

exec

$! | X T

```
exec PATHNAME LIST
exec LIST
```

Функция `exec` прекращает выполнение текущей программы, выполняет внешнюю команду и *никогда не возвращает управление!!!* Используйте `system`, если требуется восстановить управление после выхода из команды. Вызов функции `exec` завершается неудачей и возвращает ложное значение, только если команда не существует *и* если она выполняется непосредственно, а не через интерпретатор команд системы (читайте далее).

Если функция получает единственный скалярный аргумент, выполняется поиск в этом аргументе метасимволов интерпретатора команд, и в случае успеха аргумент целиком передается стандартному интерпретатору команд системы (*/bin/sh* в UNIX). Если метасимволы не найдены, аргумент расщепляется на слова и выполняется непосредственно, поскольку это позволяет повысить эффективность за счет устранения накладных расходов на обработку интерпретатором команд. Это также дает бóльшие возможности управления восстановлением после ошибки, если программа не существует.

Если в *LIST* больше одного аргумента или *LIST* представляет собой массив, в котором более одного значения, системный интерпретатор команд не используется, что также исключает какую-либо обработку команды интерпретатором команд. Наличие или отсутствие метасимволов в аргументах не оказывает влияния на этот режим, что делает форму вызова со списком предпочтительной в программах, где

желательно предотвратить атаки через инъекции escape-последовательностей интерпретатора команд.

Следующий пример заставляет выполняемую в данный момент программу Perl заменить себя программой *echo*, которая выводит текущий список аргументов:

```
exec "echo", "Ваши аргументы  ", @ARGV;
```

Этот пример показывает, что через exec можно запустить не только отдельную программу, но и целый конвейер.

```
exec "sort $outfile | uniq"
    || die "Невозможно выполнить sort/uniq: $!"
```

Обычно возврат из exec не происходит, а если происходит, то возвращается ложное значение, и следует проверить $!, чтобы узнать причину. Учтите, что в прежних версиях Perl функции exec (и system) не очищали выходной буфер, поэтому требовалось включать буферизацию установкой $| для одного или более дескрипторов файлов, чтобы избежать потери выводимых данных при вызове exec или их смешивания при вызове system.

Посылая операционной системе запрос на выполнение новой программы в существующем процессе (как это делает функция exec), вы сообщаете системе местонахождение программы, которую необходимо выполнить, а также сообщаете новой программе (через первый аргумент) имя, под которым она вызвана. Обычно сообщаемое имя является просто копией адреса программы, но это не обязательно, так как существуют два различных аргумента на уровне языка С. Если это не копия, получается необычный результат: запущенная программа думает, что она выполняется под определенным именем, которое может отличаться от того пути, где фактически находится программа. Часто это не имеет значения для программы, но иногда программы обращают на это внимание и «надевают» разную личину, в зависимости от своего текущего имени. Например, редактор *vi* проверяет, вызвали его как "vi" или как "view". Будучи назван "view", он автоматически включает режим «только для чтения», как если бы был вызван с параметром командной строки *-R*.

Здесь вступает в игру *PATHNAME* – необязательный параметр exec. Синтаксически он занимает позицию косвенного объекта, как дескриптор файла для print или printf. Запятая после него недопустима, поскольку этот параметр – не совсем часть списка аргументов. (В некотором смысле Perl выбирает подход, противоположный установленному операционной системой, предполагая, что важным является первый аргумент, и позволяя изменять имя пути, если оно отличается.) Например:

```
$editor = "/usr/bin/vi";
exec $editor "view", @files      # включаем режим "только чтение"
    || die "Невозможно выполнить $editor: $!";
```

Как и для любого другого косвенного объекта, простой скаляр с именем программы можно заменить блоком с произвольным кодом, что упрощает предыдущий пример:

```
exec { "/usr/bin/vi" } "view" @files    # включаем режим "только чтение"
    || die "Невозможно выполнить /usr/bin/vi: $!";
```

Мы уже говорили, что exec рассматривает разрозненный список аргументов как указание обойтись без обработки интерпретатором команд. Однако в одном случае

можно все же споткнуться. Вызов exec (и system тоже) не различает единственный скалярный аргумент и массив, состоящий из единственного элемента.

```
@args = ("echo surprise"); # всего один элемент в списке
exec @args                 # будут обрабатываться escape-последовательности
    || die "exec: $!"      # интерпретатора команд, потому что @args == 1
```

Чтобы избежать этого, можно использовать синтаксис *PATHNAME*, явно дублируя первый элемент в качестве имени пути, благодаря чему остальные аргументы будут интерпретироваться как список, даже если аргумент всего один:

```
exec { $args[0] } @args    # безопасно даже со списком из одного аргумента
    || die "can't exec @args: $!";
```

Первая версия, без фигурных скобок, запустит программу *echo* со строкой "surprise" в качестве аргумента. Вторая версия поступит иначе: она попытается запустить программу, буквально названную *echo surprise*, не найдет ее (хочется верить) и установит признак ошибки в переменной $!.

Поскольку функция exec чаще всего используется сразу после fork, предполагается, что все, обычно происходящее при завершении процесса Perl, должно быть пропущено. После вызова exec Perl не станет вызывать блоки END или какие-либо методы DESTROY, связанные с какими бы то ни было объектами. Напротив, порожденный процесс, в итоге, будет вынужден провести ту уборку, которую предположительно должен был бы сделать родительский процесс. (Неплохо бы так в реальной жизни.)

Поскольку очень часто exec ошибочно используют вместо system, Perl выводит предупреждение, если следующая команда отлична от die, warn или exit и включен вывод предупреждений. Если действительно необходимо поместить после exec какую-то другую команду, избежать вывода предупреждения можно любым из следующих способов:

```
exec ("foo")   || print STDERR "невозможно exec foo: $!";
{ exec ("foo") }; print STDERR "невозможно exec foo: $!";
```

Как показывает вторая строчка, вызов exec в качестве последней команды блока избавляет нас от этого предупреждения.

Перед выполнением exec Perl пытается вытолкнуть буферы всех файлов, открытых для вывода, но на некоторых платформах это может быть невозможно. Для большей безопасности, чтобы избежать потери выводимых данных, можно установить $| ($AUTOFLUSH в модуле English) или вызвать метод autoflush из модуля IO::Handle для всех открытых дескрипторов файлов.

Обратите внимание, что exec не выполняет блоки END и не вызывает методы DESTROY ваших объектов.

См. также system.

exists

```
exists EXPR
```

Возвращает истинное значение, если указанный ключ хеша существует в хеше, даже если его значение не определено.

```
print "True\n"    if        $hash{$key};
print "Exists\n"  if exists  $hash{$key};
print "Defined\n" if defined $hash{$key};
```

Исторически сложилось так, что функцию exists можно вызывать и для элементов массива, но в этом случае ее поведение не так очевидно и сильно зависит от особенностей работы функции delete с массивами. Однако в настоящее время не рекомендуется применять exists к массивам, и данная возможность, вероятно, исчезнет в будущих версиях Perl.

```
print "True\n"    if        $array[$index];
print "Exists\n"  if exists  $array[$index];
print "Defined\n" if defined $array[$index];
```

Элемент может быть истинным, только если определен, и может быть определен, только если существует, но обратное утверждение не всегда верно.

Выражение *EXPR* может иметь произвольную сложность – при условии, что последней операцией выражения будет поиск ключа в хеше или взятие из массива по адресу:

```
if (exists $hash{A}{B}{$key}) {     }
```

Хотя последний элемент не начинает существовать только потому, что было проверено его существование, к промежуточным элементам это относится. Поэтому элементы $$hash{"A"} и $hash{"A"}->{"B"} появятся сами по себе. Это не функция exists как таковая; это происходит везде, где используется оператор стрелки (явно или косвенно):

```
undef $ref;
if (exists $ref->{"Some key"}) { }
print $ref;   # выведет HASH(0x80d3d5c)
```

Здесь элемент "Some key" не возник из ниоткуда, но ранее неопределенная переменная $ref начинает вдруг содержать анонимный хеш. Это удивительный пример ***самооживления*** там, где на первый (и даже на второй) взгляд нет контекста левой части присваивания. Такое поведение, вероятно, будет исправлено в будущих версиях. Чтобы обойти это, можно использовать вложенные вызовы:

```
if (       $ref                     &&
    exists $ref->[$x]               &&
    exists $ref->[$x][$y]           &&
    exists $ref->[$x][$y]{$key}     &&
    exists $ref->[$x][$y]{$key}[2] ) {     }
```

Если *EXPR* является именем подпрограммы, функция exists возвращает истинное значение, если эта подпрограмма объявлена – даже если она еще не определена. Следующий код выводит только "Exists":

```
sub flub;
print "Exists\n"  if exists  &flub;
print "Defined\n" if defined &flub;
```

Вызов exists с именем подпрограммы может пригодиться в подпрограмме AUTOLOAD, если потребуется узнать, нужно ли определить некоторую подпрограмму в данном пакете. Пакет может указать на это, объявив подпрограмму-заглушку при помощи sub, как это было сделано для flub в примере выше.

Имейте в виду, что использование значения, возвращаемого *вызовом* подпрограммы, вместо *имени* подпрограммы, вызовет ошибку в exists.

```
exists &sub;   # OK
exists &sub(); # Ошибка  круглые скобки означают вызов функции
```

exit

```
exit EXPR
exit
```

Вычисляет *EXPR* как целое число и немедленно завершает программу с этим значением в качестве кода ошибки. Если выражение *EXPR* опущено, функция завершает программу с кодом 0 (означающим отсутствие ошибки). Вот фрагмент, позволяющий пользователю выйти из программы, введя x или X:

```
$ans = <STDIN>;
exit if $ans =~ /^[Xx]/,
```

Не следует применять exit для прерывания выполнения подпрограммы, если существует вероятность, что кто-нибудь захочет перехватывать возникшие ошибки. Вызывайте вместо нее die, которую можно перехватывать с помощью eval. Либо используйте обертки для die из модуля Carp, например croak или confess.

Мы сказали, что функция exit осуществляет немедленный выход, но это была наглая ложь. Она выходит, как только это станет возможным, но сначала вызовет все имеющиеся подпрограммы END для выполнения заключительных операций. Эти подпрограммы не могут прервать выход, но могут изменить итоговое значение кода завершения, установив значение переменной $?. Аналогично любой класс, в котором определен метод DESTROY, вызовет его для всех своих объектов, прежде чем произойдет окончательный выход из программы. Если действительно требуется обойти обработку при выходе, можно вызвать функцию _exit из модуля POSIX; это позволит избежать обработки всех END и деструкторов. А если POSIX отсутствует, можно вызвать exec "/bin/false" или сделать что-либо аналогичное.

exp




```
exp EXPR
exp
```

Возвращает *e* (основание натурального логарифма) в степени *EXPR*. Чтобы получить значение *e*, выполните exp(1). Для возведения в степень по любому основанию используйте оператор **, украденный нами из FORTRAN:

```
use Math::Complex,
print -exp(1) ** (i * pi); # выведет 1
```

__FILE__

Специальная лексема, возвращающая имя файла, в котором произошло обращение к ней. См. раздел «Генерирование Perl в других языках» в главе 21.

fc

$_ T

```
fc EXPR
fc
```

Эта функция, впервые появившаяся в Perl версии v5.16 и доступная в области действия прагмы `use feature "fc"`, возвращает полную свертку регистра символов Юникода в *EXPR*. Это – внутренняя функция, реализующая действие escape-последовательности \F для свертки регистра в строках. Как заглавный регистр основан на верхнем регистре, но несколько отличается, так и свернутый регистр основан на нижнем регистре, но отличается. В наборе символов ASCII имеется всего два регистра символов, однозначно соответствующих один другому, но в Юникоде используется отношение «один-ко-многим», и уже между тремя регистрами. Из-за того, что каждый раз возникает слишком много комбинаций, требующих проверки вручную, был введен четвертый регистр, свернутый (foldcase), используемый как промежуточный для всех трех. Это не регистр символов в полном смысле слова, но он распознается.

Чтобы сравнить две строки без учета регистра символов, выполните следующее:

```
fc($a) eq fc($b)
```

До версии v5.16 единственным надежным способом сравнения строк без учета регистра символов был модификатор шаблонов `/i`, потому что механизм поиска по шаблону в Perl всегда использовал семантику свертки регистра при поиске соответствий без учета регистра. Зная это, можно имитировать операцию определения равенства, как показано ниже:

```
sub fc_eq($$) {
    my($a, $b) = @_;
    return $a =~ /\A\Q$b\E\z/i;
}
```

В более ранних версиях, предшествующих v5.16, функцию `fc` можно найти в модуле `Unicode::CaseFold` из CPAN. Для сравнения без учета регистра и акцентов можно использовать методы `eq` и `cmp` объекта `Unicode::Collate`, которому в конструкторе было передано значение `level=>1`. Или с помощью объекта `Unicode::Collate::Locale`, сконструированного аналогичным образом и предназначенного для сопоставления строк с учетом особенностей национальных алфавитов. См. разделы «Ошибочные представления о регистре» и «Сравнение и сортировка строк Юникода» в главе 6.

fcntl




```
fcntl FILEHANDLE, FUNCTION, SCALAR
```

Вызывает функции управления файлами операционной системы, документированные на странице руководства *fcntl*(2). Перед вызовом `fcntl` необходимо использовать директиву:

```
use Fcntl;
```

чтобы загрузить определения констант.

Аргумент *SCALAR* может служить как входным, так и выходным значением (или и тем, и другим) в зависимости от *FUNCTION*. Указатель на строковое значение *SCALAR* будет передан в фактический вызов *fcntl* в третьем аргументе. (Если *SCALAR* не имеет строкового значения, но имеет числовое значение, будет передано непосредственно это значение, а не указатель на строковое значение.) Наиболее употребительные допустимые значения для *FUNCTION* вы найдете в описании модуля Fcntl.

В системах, не поддерживающих *fcntl*(2), вызов функции fcntl вызовет исключение. В системах, где такая поддержка имеется, можно, например, модифицировать флаги закрытия при exec (если нежелательно использовать переменную $^F ($SYSTEM_FD_MAX)) и флаги неблокирующего ввода/вывода, эмулировать функцию *lockf*(3) и организовывать прием сигнала SIGIO при появлении событий ввода/вывода, ожидающих обработки.

Следующий пример демонстрирует перевод дескриптора файла с именем REMOTE в неблокирующий режим на системном уровне. В результате любая операция ввода немедленно возвращает управление, если нет ничего, что можно было бы прочитать из канала, сокета или последовательной линии, которая иначе была бы заблокирована. Кроме того, в этом режиме операции вывода не блокируются, а возвращают признак неудачи. (В этом случае вам, вероятно, придется также уладить дело с $|.)

```
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);

$flags = fcntl(REMOTE, F_GETFL, 0)
            || die "Невозможно получить флаги для сокета: $!"

$flags = fcntl(REMOTE, F_SETFL, $flags | O_NONBLOCK)
            || die "Невозможно установить флаги для сокета: $!";
```

Значения, возвращаемые fcntl (и ioctl), перечислены в табл. 27.1.

Таблица 27.1. Значения, возвращаемые функцией fcntl

Значения, возвращаемые системным вызовом	Значения, возвращаемые Perl
-1	undef
0	строка "0 but true"
все прочее	это же число

Таким образом, Perl возвращает истинное значение при успехе и ложное при неудаче, хотя сохраняется возможность определить фактическое значение, возвращенное операционной системой:

```
$retval = fcntl(...) || -1;
printf "fcntl фактически вернула %d\n", $retval;
```

Здесь даже строка "0 but true" выводится как 0 благодаря формату %d. Эта строка истинна в логическом контексте и принимает значение 0 в числовом контексте. Допустимо также использовать простую проверку || die возвращаемого значения вместо «перекошенной» версии // die. (Она также благополучно избавляет от обычных предупреждений о недопустимом числовом преобразовании.)

fileno



```
fileno FILEHANDLE
```

Возвращает указатель файла в основе дескриптора файла. Если дескриптор не был открыт, fileno возвращает undef. Если дескриптор файла не связан с действительным указателем файла операционной системы (что характерно для дескрипторов файлов, связанных с объектами памяти, открытыми вызовом open со ссылкой в третьем аргументе), возвращается -1.

Указатель файла – это маленькое неотрицательное целое число, например 0 или 1. Сравните с дескрипторами файлов – например, STDIN и STDOUT, которые представляют собой символы. К несчастью, операционной системе ничего не известно о ваших замечательных символах. Она представляет открытые файлы только как эти маленькие номера файлов, и хотя Perl обычно автоматически выполняет трансляцию, иногда требуется знать фактический указатель файла.

Функция fileno может пригодиться, например, при создании битовых массивов для select и для передачи некоторым системным вызовам посредством *syscall*(2). С ее помощью также можно дополнительно проверить, вернула ли функция open нужный указатель файла, и определить, не используется ли один и тот же системный указатель файла двумя дескрипторами.

```
if (fileno(THIS) == fileno(THAT)) {
    say "THIS и THAT являются дубликатами";
}
```

Если *FILEHANDLE* является выражением, его значение принимается в качестве косвенного дескриптора файла, обычно его имени или чего-то похожего на объект дескриптора файла.

Не полагайтесь на сохранение связи между дескриптором файла Perl и числовым указателем файла на протяжении всего времени работы программы. Если файл закрывается и открывается заново, указатель может измениться. Perl берет на себя некоторый труд, стараясь, чтобы некоторые указатели не потерялись, если вызов open для них окажется неудачным, но делает это только для указателей файлов, не выше текущего значения специальной переменной $^F ($SYSTEM_FD_MAX), по умолчанию равного 2. Хотя дескрипторам файлов STDIN, STDOUT и STDERR исходно сопоставлены указатели файлов 0, 1 и 2 (стандартное соглашение UNIX), даже они могут измениться, если начать закрывать и открывать их достаточно энергично. Неприятностей с 0, 1 и 2 быть не может, если открывать их заново сразу после закрытия. Основное правило в UNIX – выбирать наименьший свободный указатель, а это именно тот, который был только что закрыт.

flock



```
flock FILEHANDLE, OPERATION
```

Является переносимым интерфейсом блокировки файлов. Блокирует только целые файлы, а не отдельные записи. Эта функция управляет блокировкой файла, связанного с *FILEHANDLE*, возвращая истинное значение в случае успеха и ложное в противном случае. Чтобы исключить возможность потери данных, Perl очищает буфер *FILEHANDLE* перед блокированием или разблокированием. Функцию flock можно реализовать посредством *flock*(2), *fcntl*(2), *lockf*(3) или другого специфи-

ческого для платформы механизма блокировки, но если ни один из них не доступен, вызов flock возбуждает исключение. См. раздел «Блокировка файлов» главы 15.

Аргумент *OPERATION* может принимать значения LOCK_SH, LOCK_EX или LOCK_UN, которые можно объединять через ИЛИ с LOCK_NB. Эти константы традиционно имеют значения 1, 2, 8 и 4, но можно обращаться к ним по символическим именам, если импортировать их из модуля Fcntl по отдельности или целиком, используя тег :flock.

LOCK_SH запрашивает блокировку с совместным доступом, поэтому обычно она применяется для чтения. LOCK_EX запрашивает монопольную блокировку, поэтому обычно используется для записи. LOCK_UN освобождает ранее запрошенную блокировку; закрытие файла тоже снимает все блокировки. Если совместно с LOCK_SH или LOCK_EX используется бит LOCK_NB, вызов flock возвращает управление немедленно, не ожидая, пока недоступная блокировка освободится. Проверяйте возвращаемое значение, чтобы определить, получена ли запрошенная блокировка. Будучи вызванной без флага LOCK_NB, flock может заблокировать выполнение программы навечно, ожидая освобождения блокировки.

Еще одна неочевидная, но традиционная особенность flock заключается в *рекомендательном характере ее блокировок*. Такие блокировки обладают большей гибкостью, но дают меньше гарантий, чем жесткие блокировки. Это значит, что файлы, заблокированные вызовом flock, могут быть модифицированы программами, не использующими flock. Машины, останавливающиеся на красный свет, хорошо уживаются друг с другом, но не с машинами, которые не останавливаются на красный свет. Будьте внимательны за рулем.

Некоторые реализации flock не могут блокировать файлы в сетевой среде. Хотя теоретически можно применять с этой целью более специфическую для системы функцию fcntl, но присяжные (удалившись на совещание по этому делу лет десять назад) все еще не решили, надежно ли это (и даже может ли это быть надежным в принципе).

Следующая программа добавляет корреспонденцию в почтовый ящик в системах UNIX, используя *flock*(2) для блокировки почтовых ящиков:

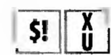
```
use Fcntl qw/:flock/;          # импорт констант LOCK_*
sub mylock {
    flock(MBOX, LOCK_EX)
        || die "невозможно блокировать mailbox: $!",
    # на случай, если кто-то что-то дописал, пока мы ждали,
    # и наш буфер stdio рассинхронизировался
    seek(MBOX, 0, 2)
        || die "невозможно seek в конец mailbox: $!";
}

open(MBOX, ">> /usr/spool/mail/$ENV{USER}")
    || die "невозможно открыть mailbox: $!";

mylock();
say MBOX $msg, "\n";
close MBOX
    || die "невозможно закрыть mailbox. $!";
```

В системах, поддерживающих системный вызов *flock*(2), блокировки наследуются процессами-потомками, порожденными вызовом fork. Другим реализациям не так повезло, и при ветвлении они, скорее всего, теряют блокировки. См. также в разделе «Блокировка файлов» главы 15 другие примеры использования flock.

fork



```
fork
```

Делает из одного процесса два посредством системного вызова *fork*(2). В случае успеха возвращает ID нового порожденного процесса в родительский процесс и 0 в порожденный процесс. Если у системы недостаточно ресурсов для размещения нового процесса, вызов оказывается неудачным, и возвращается undef. Указатели файлов (иногда даже с блокировками) используются совместно, а все остальное копируется (или, по крайней мере, такое создается впечатление).

В старых версиях Perl неочищенные буферы остаются неочищенными в обоих процессах, что означает необходимость установить $| для одного или более файловых дескрипторов ранее в программе, чтобы избежать дублирования вывода.

Почти безопасный способ породить процесс, обращая внимание на возможные ошибки ветвления, может выглядеть так:

```
use Errno qw(EAGAIN);
FORK: {
    if ($pid = fork) {
        # родительский процесс
        # pid порожденного процесса в $pid
    }
    elsif (defined $pid) { # $pid равен нулю, если определен
        # порожденный процесс
        # pid родительского процесса доступен через getppid
    }
    elsif ($! == EAGAIN) {
        # EAGAIN является предположительно нефатальной ошибкой ветвления
        sleep 5;
        redo FORK;
    }
    else {
        # фатальная ошибка ветвления
        die "Ветвление процессов невозможно: $!";
    }
}
```

Такие предосторожности не обязательны для операций, которые неявно вызывают *fork*(2), например, system, обратные апострофы или открытие процесса как дескриптора файла, потому что Perl, вызывая fork за вас, автоматически повторяет попытки породить процесс, не обращая внимания на временные затруднения. Не забывайте заканчивать работу в порожденном процессе вызовом exit, иначе потомок выйдет из блока условной инструкции и начнет выполнять код, предназначенный только для родительского процесса.

Вызывая fork и не дожидаясь завершения порожденных процессов, вы накопите «зомби» (процессы, завершившиеся против ожидания родителей). В некоторых системах этого можно избежать, установив $SIG{CHLD} в "IGNORE"; в большинстве слу-

чаев нужно вызывать wait для умирающих дочерних процессов. Примеры см. в описании функции wait, а подробности о SIGCHLD – в разделе «Сигналы» главы 15.

Если ответвившийся процесс наследует дескрипторы системных файлов, такие как STDIN и STDOUT, соединенные с удаленным каналом или сокетом, удаленный сервер (например, сценарий CGI или фоновое задание, запущенное в удаленном интерпретаторе команд) окажется зависшим, даже если завершится родительский процесс. Повторное соединение системных дескрипторов с другими выходами исправит положение.

В большинстве систем, поддерживающих *fork*(2), эта функция имеет высокоэффективную реализацию (например, использование технологии копирования при записи (copy-on-write) для страниц данных), что обусловило доминирование этой парадигмы многозадачности в последние десятилетия. Трудно рассчитывать, что функция fork реализована эффективно (если реализована вообще) в не-UNIX системах. Например, Perl эмулирует правильную fork даже в системах Microsoft, но в этом случае ничего нельзя утверждать относительно производительности. Возможно, большая удача ждет вас в случае применения модуля Win32::Process.

Перед созданием дочернего процесса Perl пытается вытолкнуть буферы файлов, открытых для вывода, но на некоторых платформах это может быть невозможно. Чтобы избежать двукратного вывода одних и тех же данных, можно установить $| ($AUTOFLUSH в модуле English) или вызвать метод autoflush из модуля IO::Handle для открытых дескрипторов файлов.

format

```
format NAME =
    picture line
    value list
    ...
.
```

Объявляет именованную последовательность строк шаблона (picture lines) и связанных величин для функции write. Если аргумент *NAME* опущен, по умолчанию выбирается имя STDOUT, которое также является именем формата по умолчанию для дескриптора файла STDOUT. Поскольку это глобальное для пакета объявление, обрабатываемое на этапе компиляции, все переменные в списке значений должны быть видимы в точке объявления формата, т. е. лексические переменные должны быть объявлены выше в файле, а переменные с динамической областью видимости должны быть установлены во время вызова write. Например (предполагается, что значения $cost и $quantity уже вычислены):

```
my $str = "widget";     # Переменная с лексической областью видимости

format Nice_Output =
Test: @<<<<<<<< @||||| @>>>>>
      $str,     $%,    '$'   int($num)
.

local $~ = "Nice_Output";       # Выбираем наш формат
local $num = $cost * $quantity; # Переменная с динамической областью видимости

write;
```

Подобно файловым дескрипторам, имена форматов являются идентификаторами, существующими в таблице символов (в пакете), и могут квалифицироваться

именем пакета. Внутри таблицы символов форматы располагаются в собственном пространстве имен, отдельно от дескрипторов файлов, дескрипторов каталогов, скаляров, массивов, хешей и подпрограмм. Однако, как и эти шесть типов, формат с именем Whatever подвержен воздействию директивы local, при применении ее к *Whatever. Иными словами, формат – это еще одна штуковина, независимая от других штуковин и содержащаяся в переменной typeglob.

Раздел «Форматы шаблонов» главы 26 содержит многочисленные детали и примеры применения форматов. Глава 25 описывает внутренние специфические для форматов переменные, а модули English и IO::Handle предоставляют более легкий доступ к ним.

formline

```
formline PICTURE, LIST
```

Внутренняя функция, которую вызывает format, хотя можно вызвать ее и самостоятельно. Всегда возвращает истинное значение. formline форматирует список значений согласно содержимому шаблона *PICTURE*, помещая вывод в накопитель форматированного вывода, $^A (или $ACCUMULATOR, если используется модуль English). В конечном счете, вызов write выводит содержимое $^A в какой-нибудь дескриптор файла, но можно прочесть $^A самостоятельно, а затем записать в $^A пустую строку "". Обычно formline вызывается для каждой строки формата, но сама функция formline не смотрит, сколько символов перевода строки имеется в *PICTURE*. Это значит, что маркеры ~ и ~~ будут интерпретировать *PICTURE* как одну строку. Поэтому может потребоваться несколько вызовов formline для реализации формата одной записи, как это и делает компилятор формата.

Будьте осторожны, если заключаете шаблон в двойные кавычки, поскольку символ @ может быть принят за начало имени массива. Примеры использования см. в разделе «Форматы шаблонов» главы 26.

getc




```
getc FILEHANDLE
getc
```

Возвращает очередной байт из входного файла, прикрепленного к *FILEHANDLE*. В позиции конца файла или при возникновении ошибки ввода/вывода возвращает undef. Если дескриптор *FILEHANDLE* опущен, чтение производится из STDIN.

Эта функция несколько медлительна, но иногда полезна для ввода одиночных символов с клавиатуры – при условии, что ввод с клавиатуры не буферизован. Эта функция запрашивает небуферизованный ввод из стандартной библиотеки ввода/вывода. К несчастью, стандартная библиотека ввода/вывода не настолько стандартна, чтобы обеспечить переносимый способ получить от операционной системы небуферизованный ввод с клавиатуры для стандартной системы ввода/вывода. Для этого придется проявить сообразительность, учесть особенности разных операционных систем. В UNIX можно поступить так:

```
if ($BSD_STYLE) {
    system "stty cbreak </dev/tty >/dev/tty 2>&1";
} else {
    system "stty", "-icanon", "eol", "\001",
}
```

```
$key = getc,

if ($BSD_STYLE) {
    system "stty -cbreak </dev/tty >/dev/tty 2>&1";
} else {
    system "stty", "icanon", "eol", "^@"; # ASCII NUL
}
print "\n";
```

Этот код помещает очередной символ, введенный с терминала, в строку $key. Если программа *stty* поддерживает такой параметр, как cbreak, потребуется использовать код, соответствующий истинному значению $BSD_STYLE. В противном случае придется использовать код, соответствующий ложному значению. Определение параметров *stty*(1) мы оставляем читателям в качестве упражнения.

Модуль POSIX предоставляет более переносимый вариант такой реализации посредством функции POSIX::getattr. Более переносимый и гибкий подход можно также найти в модуле Term::ReadKey на ближайшем к вам сайте CPAN. Вместо функции ungetc можно использовать метод класса IO::Handle.

getgrent




```
getgrent
setgrent
endgrent
```

Эти подпрограммы выполняют обход содержимого вашего (или чужого, если он находится где-то на сервере) файла */etc/group*. В списочном контексте getgrent возвращает:

```
# 0      1        2     3
($name, $passwd, $gid, $members) = getgrent();
```

где $members содержит список разделенных пробелами имен пользователей, членов группы. Организовать хеш для перевода имен групп в GID можно так:

```
while (($name, $passwd, $gid) = getgrent) {
    $gid{$name} = $gid;
}
```

В скалярном контексте getgrent возвращает только имя группы. Стандартный модуль User::grent поддерживает интерфейс к этой функции через имена. См. *getgrent*(3).

getgrgid




```
getgrgid GID
```

Ищет запись в файле групп по номеру группы. В списочном контексте возвращает:

```
# 0      1        2     3
($name, $passwd, $gid, $members)
    = getgrgid(0);
```

где $members содержит список разделенных пробелами имен пользователей, членов группы. Если это необходимо делать неоднократно, подумайте о кэшировании данных в хеше с помощью getgrent.

В скалярном контексте getgrgid возвращает только имя группы. Модуль User::grent поддерживает интерфейс к этой функции через имена. См. *getgrgid*(3).

getgrnam



```
getgrnam NAME
```

Ищет запись в файле групп по имени группы. В списочном контексте возвращает:

```
# 0      1        2     3
($name, $passwd, $gid, $members) =
    getgrnam("wheel");
```

где $members содержит список разделенных пробелами имен пользователей, членов группы. Если приходится делать это неоднократно, подумайте о кэшировании данных в хеше с помощью getgrent.

В скалярном контексте getgrnam возвращает только числовой идентификатор группы. Модуль User::grent поддерживает интерфейс к этой функции через имена. См. *getgrnam*(3).

gethostbyaddr



```
gethostbyaddr ADDR, ADDRTYPE
```

Транслирует адреса в имена (и в альтернативные адреса). *ADDR* должен быть упакованным двоичным сетевым адресом, а в аргументе *ADDRTYPE* на практике обычно передается AF_INET (из модуля Socket). В списочном контексте возвращает:

```
# 0      1         2          3        4 ..
($name, $aliases, $addrtype, $length, @addrs) =
    gethostbyaddr($packed_binary_address, $addrtype);
```

где @addrs является списком упакованных двоичных адресов. В домене Интернета каждый адрес (исторически) имеет длину четыре байта и может быть распакован, например, так:

```
($a, $b, $c, $d) = unpack("C4", $addrs[0]);
```

Альтернативный вариант: прямое преобразование в нотацию вектора с точками с помощью модификатора v для sprintf:

```
$dots = sprintf "%vd", $addrs[0];
```

Функцию inet_ntoa из модуля Socket удобно использовать при создании версии для вывода через print. Этот подход станет важным, если мы все когда-либо перейдем на IPv6.

```
use Socket;
$printable_address = inet_ntoa($addrs[0]);
```

В скалярном контексте gethostbyaddr возвращает только имя хоста.

Чтобы создать *ADDR* из вектора с точками, сделайте так:

```
use Socket;
$ipaddr = inet_aton("127.0.0.1");       # localhost
$claimed_hostname = gethostbyaddr($ipaddr, AF_INET);
```

Дополнительные примеры вы найдете в разделе «Сокеты» главы 15. Модуль Net::hostent обеспечивает поддержку интерфейса к этой функции через имена. См. *gethostbyaddr*(3).

Модуль Socket содержит функцию gethostinfo, поддерживающую адреса во всех распространенных форматах, включая IPv6.

Функция getaddrinfo используется для получения списка IP-адресов и номеров портов для указанного хоста (и, возможно, имени службы), и обеспечивает бóльшую гибкость, чем функции *gethostbyname*(3) и *getservbyname*(3).

```
use Socket qw(:all);
@addr_structs = getaddrinfo("127.0.0.1");   # петлевой адрес IPv4
@addr_structs = getaddrinfo("207.171.7.72");

@addr_structs = getaddrinfo("::1");         # петлевой адрес IPv6
@addr_structs = getaddrinfo("e80::223:12ff:fe58:714c");
```

gethostbyname



```
gethostbyname NAME
```

Транслирует сетевое имя хоста в соответствующие адреса (и другие имена). В списочном контексте возвращает значение:

```
# 0      1         2          3       4 ..
($name, $aliases, $addrtype, $length, @addrs) =
gethostbyname($remote_hostname);
```

где @addrs – список «сырых» адресов. В домене Интернета каждый адрес (исторически) имеет длину четыре байта и может быть распакован, например, так:

```
($a, $b, $c, $d) = unpack("C4", $addrs[0]);
```

Адрес можно непосредственно преобразовать в нотацию вектора с точками с помощью модификатора v для sprintf:

```
$dots = sprintf "%vd", $addrs[0];
```

В скалярном контексте gethostbyname возвращает только адрес хоста:

```
use Socket;
$ipaddr = gethostbyname($remote_host);
printf "%s has address %s\n",
    $remote_host, inet_ntoa($ipaddr),
```

О другом подходе рассказано в разделе «Сокеты» главы 15. Модуль Net::hostent обеспечивает поддержку интерфейса к этой функции через имена. См. также *gethostbyname*(3).

gethostent



```
gethostent
sethostent STAYOPEN
endhostent
```

Эти функции осуществляют итерации по файлу */etc/hosts* и возвращают каждый раз по одной записи. Функция gethostent возвращает значение:

```
($name, $aliases, $addrtype, $length, @addrs)
```

где @addrs является списком «сырых» адресов. В домене Интернета каждый адрес (исторически) имеет длину четыре байта и может быть распакован, например, так:

```
($a, $b, $c, $d) = unpack("C4", $addrs[0]);
```

Сценарии, содержащие вызов gethostent, не следует считать переносимыми. Если система использует сервер имен, ей придется опросить большую часть Интернета, чтобы удовлетворить запрос всех адресов всех компьютеров на свете. Поэтому gethostent не реализована в таких системах. Другие подробности можно найти в описании *gethostent*(3).

Модуль Net::hostent обеспечивает поддержку интерфейса к этой функции через имена.

getlogin



```
getlogin
```

Возвращает имя текущего пользователя, если оно найдено. В UNIX-системах оно извлекается из файла *utmp*(5). Если возвращает ложное значение, используйте getpwuid. Например:

```
$login = getlogin() || (getpwuid($<))[0] || "Нарушитель!!"
```

getnetbyaddr



getnetbyaddr *ADDR, ADDRTYPE*

Транслирует сетевой адрес в соответствующее сетевое имя или имена. В списочном контексте возвращает значение:

```
use Socket;
($name, $aliases, $addrtype, $net) = getnetbyaddr(127, AF_INET);
```

В скалярном контексте возвращает сетевое имя. Модуль Net::netent обеспечивает поддержку интерфейса к этой функции через имена. См. *getnetbyaddr*(3).

getnetbyname



getnetbyname *NAME*

Транслирует сетевое имя в соответствующий сетевой адрес. В списочном контексте возвращает значение:

```
($name, $aliases, $addrtype, $net) = getnetbyname("loopback");
```

В скалярном контексте возвращает только сетевой адрес. Модуль Net::netent обеспечивает поддержку интерфейса к этой функции через имена. См. *getnetbyname*(3).

getnetent



getnetent
setnetent *STAYOPEN*
endnetent

Эти функции осуществляют перебор записей файла */etc/networks*. В списочном контексте возвращают значение:

```
($name, $aliases, $addrtype, $net) = getnetent()
```

В скалярном контексте getnetent возвращает только сетевое имя. Модуль Net::netent обеспечивает поддержку интерфейса к этой функции через имена. См. *getnetent*(3).

Концепция сетевых имен кажется в наше время довольно странной; большинство IP-адресов находится в неименованных (и неименуемых) подсетях.

getpeername



getpeername *SOCKET*

Возвращает упакованный адрес сокета противоположного конца соединения *SOCKET*. Например:

```
use Socket;
$hersockaddr      = getpeername SOCK;
($port, $heraddr) = sockaddr_in($hersockaddr);
$herhostname      = gethostbyaddr($heraddr, AF_INET);
$herstraddr       = inet_ntoa($heraddr);
```

getpgrp



getpgrp *PID*

Возвращает текущую группу процессов для указанного *PID* (для текущего процесса используйте *PID*, равный 0). Вызов getpgrp вызывает исключение в системах, где не реализована *getpgrp*(2). Если параметр *PID* опущен, функция возвращает группу процессов текущего процесса (то же, что при *PID*, равном 0). В системах, реализующих этот оператор через системный вызов POSIX *getpgrp*(2), *PID* следует опускать или назначать ему нулевое значение.

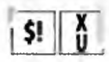
getppid



getppid

Возвращает идентификатор родительского процесса. В типичной системе UNIX, когда идентификатор родительского процесса изменяется и принимает значение 1, это означает, что первоначальный родительский процесс завершился, и родителем стала программа *init*(8).

getpriority



getpriority *WHICH, WHO*

Возвращает текущий приоритет процесса, группы процессов или пользователя. См. *getpriority*(2). Вызов getpriority вызывает исключение в системах, где не реализована *getpriority*(2).

Модуль BSD::Resource в CPAN предоставляет более удобный интерфейс, в том числе символические константы PRIO_PROCESS, PRIO_PGRP и PRIO_USER для передачи в аргументе *WHICH*. Хотя обычно они равны 0, 1 и 2 соответственно, никогда не знаешь, что произойдет в темных недрах #include-файлов стандартной библиотеки языка C.

Значение 0 в аргументе *WHO* означает текущий процесс, группу процессов или пользователя, поэтому получить приоритет текущего процесса можно так:

```
$curprio = getpriority(0, 0);
```

getprotobyname



```
getprotobyname NAME
```

Транслирует имя протокола в соответствующий номер протокола. В списочном контексте возвращает следующее значение:

```
($name, $aliases, $protocol_number) = getprotobyname("tcp")
```

В скалярном контексте возвращает только номер протокола. Модуль Net::proto поддерживает интерфейс к этой функции через имена. См. *getprotobyname*(3).

getprotobynumber



```
getprotobynumber NUMBER
```

Транслирует номер протокола в соответствующее имя протокола. В списочном контексте возвращает следующее значение:

```
#  0       1          2
($name, $aliases, $protocol_number) = getprotobynumber(6);
```

В скалярном контексте возвращает только имя протокола. Модуль Net::proto поддерживает интерфейс к этой функции через имена. См. *getprotobynumber* (3).

getprotoent



```
getprotoent
setprotoent STAYOPEN
endprotoent
```

Эти функции осуществляют перебор записей файла */etc/protocols*. В списочном контексте getprotoent возвращает значение:

```
#  0       1          2
($name, $aliases, $protocol_number) = getprotoent();
```

В скалярном контексте getprotoent возвращает только имя протокола. Модуль Net::proto поддерживает интерфейс к этой функции через имена. См. *getprotent*(3).

getpwent



```
getpwent
setpwent
endpwent
```

Эти функции осуществляют перебор записей файла */etc/passwd*, при этом в работу может вовлекаться также файл */etc/shadow*, если функции вызываются с привилегиями суперпользователя, а в системе используется теневой файл паролей. Ситуация может осложниться при использовании системы регистрации, основанной на базе данных или сетевых ресурсах. В списочном контексте возвращает следующее значение:

```
# 0     1        2    3    4      5         6     7    8
($name,$passwd,$uid,$gid,$quota,$comment,$gcos,$dir,$shell) = getpwent();
```

В некоторых системах поля квоты и комментария ($quota и $comment) могут использоваться с другими целями, чем указывают их имена, но остальные поля всегда имеют ожидаемые значения. Чтобы создать хеш для трансляции имен пользователей в UID, сделайте так:

```
while (($name, $passwd, $uid) = getpwent()) {
    $uid{$name} = $uid;
}
```

В скалярном контексте возвращает только имя пользователя. Модуль User::pwent поддерживает интерфейс к этой функции через имена. См. *getpwent*(3).

getpwnam



```
getpwnam NAME
```

Транслирует имя пользователя в соответствующую запись файла */etc/passwd*. В списочном контексте возвращает следующее значение:

```
# 0     1        2    3    4      5         6     7    8
($name,$passwd,$uid,$gid,$quota,$comment,$gcos,$dir,$shell)
    = getpwnam("daemon");
```

В системах, поддерживающих теневые пароли, для извлечения фактического пароля нужно иметь привилегии суперпользователя. Библиотека языка С должна проверить, что имеющихся привилегий достаточно, и открыть файл */etc/shadow* (или тот, где хранятся теневые пароли). По крайней мере, так она должна работать.

Для многократного поиска можно кэшировать данные с помощью getpwent.

В скалярном контексте возвращает только числовой идентификатор пользователя. Модуль User::pwent поддерживает интерфейс к этой функции через имена. См. *getpwnam*(3) и *passwd*(5).

getpwuid



```
getpwuid UID
```

Транслирует числовой идентификатор пользователя в соответствующую запись файла */etc/passwd*. В списочном контексте возвращает следующее значение:

```
# 0     1        2    3    4      5         6     7    8
($name,$passwd,$uid,$gid,$quota,$comment,$gcos,$dir,$shell) = getpwuid(2);
```

Для многократного поиска можно кэшировать данные с помощью getpwent.

В скалярном контексте возвращает имя пользователя. Модуль User::pwent поддерживает интерфейс к этой функции через имена. См. *getpwnam*(3) и *passwd*(5).

getservbyname



```
getservbyname NAME, PROTO
```

Транслирует имя службы (порта) в соответствующий номер порта. В аргументе *PROTO* передается имя протокола; например, "tcp". В списочном контексте возвращает следующее значение:

```
# 0      1          2             3
($name, $aliases, $port_number, $protocol_name) = getservbyname("www", "tcp");
```

В скалярном контексте возвращает только номер порта службы. Модуль Net::servent поддерживает интерфейс к этой функции через имена. См. *getservbyname*(3).

getservbyport



getservbyport *PORT*, *PROTO*

Транслирует номер службы (порта) в соответствующее имя. В аргументе *PROTO* передается имя протокола: например, "tcp". В списочном контексте возвращает следующее значение:

```
# 0      1          2             3
($name, $aliases, $port_number, $protocol_name) = getservbyport(80, "tcp");
```

В скалярном контексте возвращает только имя службы. Модуль Net::servent поддерживает интерфейс к этой функции через имена. См. *getservbyport*(3).

getservent



```
getservent
setservent STAYOPEN
endservent
```

Выполняет перебор записей файла */etc/services* или эквивалентного ему. В списочном контексте возвращает следующее значение:

```
# 0      1          2             3
($name, $aliases, $port_number, $protocol_name) = getservent();
```

В скалярном контексте возвращает только номер порта службы. Модуль Net::servent поддерживает интерфейс к этой функции через имена. См. *getservent*(3).

getsockname



getsockname *SOCKET*

Возвращает упакованный адрес сокета указанного конца соединения *SOCKET*. (А почему вы до сих пор не знаете свой собственный адрес? Может быть, потому, что привязали адрес, содержащий групповые символы, к сокету сервера перед выполнением accept, и теперь хотите узнать, какой интерфейс кто-то использовал для соединения с вами. Либо сокет был передан вам родительским процессом, например *inetd*.)

```
use Socket;
# предполагается, что SOCK - это сокет установленного соединения
$mysockaddr = getsockname(SOCK);
($port, $myaddr) = sockaddr_in($mysockaddr),
$myname = gethostbyaddr($myaddr, AF_INET);
printf "I am %s [%vd]\n", $myname, $myaddr;
```

getsockopt



```
getsockopt SOCKET, LEVEL, OPTNAME
```

Возвращает значение параметра *OPTNAME* сокета *SOCKET* на указанном уровне *LEVEL*, или undef в случае ошибки. Параметры могут существовать на нескольких уровнях протокола, в зависимости от типа сокета, но как минимум они определены на высшем уровне SOL_SOCKET (определяется в модуле Socket). Чтобы запросить параметры на другом уровне, необходимо указать номер соответствующего протокола, управляющего параметром. Например, чтобы указать, что требуется получить параметр на уровне протокола TCP, в аргументе *LEVEL* следует передать номер протокола TCP, который можно получить вызовом getprotobyname.

Возвращает упакованное строковое представление запрошенного параметра сокета или, в случае ошибки, undef – и описание ошибки в $!. Содержимое упакованной строки зависит от аргументов *LEVEL* и *OPTNAME*; подробности вы найдете на странице *getsockopt*(2). Но чаще значение параметра – это целое число, которое можно распаковать с применением формата "i" (или "I").

Например, чтобы проверить, включен ли алгоритм Нейгла (Nagle) в сокете:

```
use Socket qw(:all);

# предполагается, что сокет $socket хранит указатель сокета
# для установленного соединения
$tcp = IPPROTO_TCP;
$packed = getsockopt($socket, $tcp, TCP_NODELAY)
    || die "getsockopt: $!";
$nodelay = unpack("I", $packed);
printf "Алгоритм Нейгла (Nagle) в%s.\n", $nodelay ? "ыкл." : "кл.";
```

См. дополнительные сведения в описании setsockopt.

glob



```
glob EXPR
glob
```

Возвращает список имен файлов, соответствующих шаблону *EXPR*, как это делает интерпретатор команд (shell). Это – внутренняя функция, реализующая оператор <*>.

В силу исторических причин алгоритм работы этой функции соответствует алгоритму расширения в оболочке *csh*(1), а не в оболочке Борна. Имена файлов, начинающиеся точкой (.), игнорируются, если только точка не задана в шаблоне явно. Звездочка (*) соответствует любой последовательности любых символов (в том числе отсутствию символов). Вопросительный знак (?) соответствует любому единственному символу. Последовательность в квадратных скобках ([...]) задает простой класс символов, например "[chy0-9]". Классы символов могут инвертироваться с помощью знака крышки (^), как в "*.[^oa]", что соответствует любым файлам, имена которых начинаются не с точки, в именах которых есть точка, за которой следует один символ, отличный от "a" и "o", и являющийся последним в имени. Тильда (~) расширяется в домашний каталог пользователя. Например, "~/.*rc" выберет все *rc*-файлы текущего пользователя, а "~jane/Mail/*" – все почто-

вые файлы пользователя Jane. Фигурные скобки можно использовать для перечисления альтернатив, как в случае "~/.{mail,ex,csh,twm,}rc" для получения конкретных *rc*-файлов.

Функция glob исстари использует пробелы для разделения нескольких шаблонов, например <*.c *.h>. Чтобы осуществлять поиск имен файлов, которые могут содержать пробельные символы, шаблоны необходимо заключать в кавычки. Например, чтобы найти файлы, имена которых содержат символ "e", за которым следует пробел и символ "f", можно использовать такие шаблоны:

```
@spacies = <"*e f*">;
@spacies = glob '"*e f*"';
@spacies = glob q("*e f*");
```

Если потребуется добавить в шаблон содержимое переменной, это можно сделать так:

```
@spacies = glob  "'*${var}e f*'";
@spacies = glob qq("*${var}e f*");
```

Можно также воспользоваться модулем File::Glob; более подробную информацию о нем вы найдете на страницах справочного руководства (manpage). Вызов glob или оператора <*> автоматически приводит к использованию этого модуля, поэтому, если этот модуль неожиданным образом испарится из вашей библиотеки, возникнет исключение.

При вызове функции open Perl не проводит подстановку метасимволов в именах файлов, даже тильд. Поэтому предварительно необходимо вызвать glob:

```
open(MAILRC, "~/.mailrc") # ОШИБКА: тильда попадает в интерпретатор команд
    || die "невозожно открыть ~/.mailrc: $!";
open(MAILRC, <~/.mailrc>) # сначала расширить тильду
    || die "невозможно открыть ~/.mailrc: $!";
open(MAILRC, (glob("~/.mailrc"))[0])   # то же самое, но будьте внимательны
    || die "невозможно открыть ~/.mailrc: $!"  # glob возвращает список
```

Если непустые фигурные скобки – единственные групповые символы в вызове glob, поиск имен файлов не производится, но может возвращаться множество строк. Например, следующая инструкция вернет девять строк, по одной на каждое сочетание фрукта и цвета:

```
@many = glob "{apple,tomato,cherry}={green,yellow,red}";
```

Функция glob не связана с понятием переменных typeglob в Perl, если не считать того, что в обоих случаях символ * служит для представления группы объектов.

См. также раздел «Поиск файлов по шаблону» главы 2.

gmtime

```
gmtime EXPR
gmtime
```

Преобразует значение времени, возвращаемое функцией time, в список из девяти элементов, представляющих время по Гринвичу (также известное как GMT), ныне известное как всемирное координированное время (Coordinated Universal Time, UTC). Обычно она используется так:

```
# 0    1    2     3     4    5     6     7     8
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = gmtime;
```

Если опустить аргумент *EXPR*, выполняется вызов gmtime(time()). Модуль из библиотеки Perl Time::Local содержит подпрограмму timegm, которая может преобразовать список обратно в значение времени.

Все элементы списка являются числами и берутся прямо из struct tm (это программная структура C – не волнуйтесь). В частности, это означает, что $mon имеет диапазон 0..11, причем январю соответствует номер 0, $wday имеет диапазон 0..6, а воскресенью соответствует номер 0. Несложно запомнить, какие величины имеют нумерацию с нуля, поскольку именно они всегда служат индексами массивов, содержащих названия месяцев и дней недели и также имеющих нумерацию с нуля.

Например, получить текущий месяц в Лондоне можно так:

```
$london_month = (qw(Jan Feb Mar Apr May Jun
                    Jul Aug Sep Oct Nov Dec))[(gmtime)[4]];
```

$year – это число лет, прошедших после 1900; т.е. в 2023 году $year будет иметь значение 123, а *не* просто 23. Чтобы получить четырехзначный год, просто скажите $year + 1900. Чтобы получить двузначный год (например, «01» в 2001 году), используйте вызов sprintf("%02d", $year % 100).

В скалярном контексте gmtime возвращает строку в стиле *ctime*(3), основанную на значении времени GMT. Модуль Time::gmtime поддерживает интерфейс к этой функции через имена. В описании функции POSIX::strftime изложен подход к форматированию времени, дающий большую степень контроля.

Это скалярное значение *не* зависит от национальных настроек, а является встроенным для Perl. См. также модуль Time::Local и функции *strftime*(3) и *mktime*(3) в модуле POSIX. Чтобы получить сходные строки даты, не зависящие от национальных настроек, присвойте надлежащие значения переменным среды, управляющим национальными настройками (прочитайте, пожалуйста, страницу руководства *perllocale*) и попробуйте:

```
use POSIX qw(strftime);
$now_string = strftime "%a %b %e %H:%M:%S %Y", gmtime;
```

Управляющие последовательности %a и %b, представляющие сокращенные формы дня недели и месяца года, не для всех национальных настроек будут трехсимвольными.

goto




```
goto LABEL
goto EXPR
goto &NAME
```

goto *LABEL* находит инструкцию с меткой *LABEL* и продолжает выполнение с нее. Если *LABEL* не найдена, возникает исключение. Нельзя использовать эту функцию для входа в конструкцию, требующую инициализации, например, подпрограмму или цикл foreach. Нельзя также с ее помощью входить в конструкцию, удаленную в результате оптимизации. Она пригодна для перехода почти в любую точку

в динамической области видимости[1], в том числе из подпрограмм, но для этой цели обычно лучше применять какую-либо другую конструкцию, например last или die. Автор Perl никогда не испытывал потребности в этой форме goto, когда писал на Perl (а язык C – другое дело).

Достигая еще больших вершин ортогональности (и глубин идиотизма), Perl поддерживает форму goto *EXPR*, где значением выражения *EXPR* может быть имя метки, местонахождение которой *гарантированно* не известно до этапа выполнения, поскольку метка неизвестна на этапе компиляции. Это позволяет применять вычисляемые метки в операторе goto, как в FORTRAN, хотя делать это мы не рекомендуем, если вы стремитесь повысить удобство сопровождения ваших программ:

```
goto +("FOO", "BAR", "GLARCH")[$i];
```

Стоящая особняком форма goto *&NAME* таит особое волшебство, заменяя вызов указанной подпрограммы на подпрограмму, выполняющуюся в данный момент. Эту конструкцию не стыдно использовать в подпрограммах AUTOLOAD, которым нужно загрузить другую подпрограмму, а затем представить дело так, будто в первую очередь была вызвана эта новая подпрограмма, а не первоначальная (только все изменения @_ в исходной подпрограмме распространяются в заменяющую). После goto даже caller не сможет определить, что первой была вызвана подпрограмма AUTOLOAD.

grep

```
grep EXPR, LIST
grep BLOCK LIST
```

Вычисляет *EXPR* или *BLOCK* в логическом контексте для каждого элемента *LIST*, поочередно присваивая переменной $_ значение каждого элемента, подобно конструкции foreach. В списочном контексте возвращает список элементов, для которых выражение вернуло истинное значение. (Оператор назван в честь любимой программы UNIX, извлекающей из файлов строки, соответствующие заданному шаблону. В Perl выражение часто является шаблоном, но не обязано им быть.) В скалярном контексте возвращает число элементов, для которых выражение было истинным.

Если предположить, что @all_lines содержит строки кода, следующий пример удалит строки комментариев:

```
@code_lines = grep !/^\s*#/, @all_lines;
```

Поскольку $_ представляет собой неявный псевдоним для каждого значения в списке, изменение $_ вызывает изменение элементов исходного списка. Такое свойство полезно и поддерживается, но иногда приводит к результатам, которые могут оказаться странными и неожиданными. Например:

```
@list = qw(barney fred dino wilma);
@greplist = grep { s/^[bfd]// } @list;
```

@greplist теперь содержит «arney», «red», «ino», но @list содержит «arney», «red», «ino», «wilma»! Программист, Будь Осторожен.

---

[1] Это значит, что если метка не находится в текущей подпрограмме, осуществляется обратный поиск метки в подпрограммах, вызвавших данную, что делает почти невозможным сопровождение вашей программы.

См. также map. Следующие две инструкции функционально эквивалентны:

```
@out = grep { EXPR } @in;
@out = map { EXPR ? $_ . () } @in
```

Если потребуется версия grep, производящая вычисления по короткой схеме, обратите внимание на функцию first из стандартного модуля List::Util. Вместо списка всех элементов, для которых выражение *EXPR* вернет истинное значение, она возвращает только первый такой элемент или undef, если нет ни одного элемента, соответствующего условию. Как всегда, в переменную $_ записывается каждый элемент:

```
use List::Util qw(first);

$first_over_100 = first { $_ > 100 } @list;
$first_with_foo = first { /foo/ } @list;
```

А следующая функция принимает единственный символ и сообщает, в какой версии стандарта Юникода он появился впервые:

```
use v5.14;
use List::Util qw(first);
sub getage(_) {
    my $one_char = shift;
    die unless length($one_char) == 1;
    state $ages = [reverse qw(1.1 2.0 2.1 3.0 3.1 3.2
                              4.0 4.1 5.0 5.1 5.2 6.0
                             )];
    return first { $one_char =~ /\p{Age=$_}/ } @$ages;
}
```

hex

$_

```
hex EXPR
hex
```

Интерпретирует *EXPR* как шестнадцатеричную строку и возвращает эквивалентное десятичное значение. Приставка 0x, если она есть, игнорируется. Для интерпретации строк, которые могут начинаться числовым префиксом 0, 0b или 0x, используйте oct. Следующая инструкция присвоит $number значение **4 294 906 560**:

```
$number = hex("ffff12c0");
```

Обратное преобразование можно выполнить посредством sprintf:

```
sprintf "%lx", $number; # (l - буква, а не цифра.)
```

Шестнадцатеричные строки могут представлять только целые числа. Строки, вызывающие переполнение целых, вызывают исключение.

import

```
import CLASSNAME LIST
import CLASSNAME
```

Встроенной функции import не существует. Это – обычный метод класса, определяемый (или наследуемый) модулями, которые хотят экспортировать имена в другой модуль посредством оператора use. Подробности см. в описании use.

index

```
index STR, SUBSTR, OFFSET
index STR, SUBSTR
```

Ищет одну строку в другой. Возвращает позицию первого вхождения *SUBSTR* в *STR*. Смещение *OFFSET*, если задано, определяет, сколько символов от начала пропустить, прежде чем начать поиск. Позиции нумеруются с 0. Если подстрока не найдена, функция возвращает число, на единицу меньшее основания, т.е. обычно -1. Чтобы просмотреть всю строку, можно сделать так:

```
$pos = -1;
while (($pos = index($string, $lookfor, $pos)) > -1) {
    print "Найдено в позиции $pos\n"; $pos++;
}
```

Смещения всегда выражаются в символах, доступных программисту (т.е. в кодах), а не в графемах, которые видит пользователь. Смещение выражается в байтах, только если вы уже выполнили декодирование абстрактных символов в некоторую конкретную кодировку, такую как UTF-8 или UTF-16. См. главу 6.

Работать со строками, как с последовательностями графем, а не кодов, можно с помощью методов `index`, `rindex` и `pos` из модуля `Unicode::GCString`, опубликованного в CPAN.

int

$_

```
int EXPR
int
```

Возвращает целую часть *EXPR*. Программисты на C склонны забывать о применении `int` в сочетании с делением, которое в Perl является операцией с плавающей запятой:

```
$average_age = 939/16;      # дает 58.6875 (58 в языке C)
$average_age = int 939/16; # дает 58
```

Не следует использовать эту функцию для обычного округления, потому что она усекает результат в сторону 0, а машинное представление чисел с плавающей запятой может иногда приводить к неочевидным результатам. Например, `int(-6.725/0.025)` вернет -268, а не правильное -269, так как операция деления в действительности дает что-то вроде -268.99999999999994315658. В большинстве случаев следует предпочесть функции `sprintf`, `printf` или `POSIX::floor` и `POSIX::ceil`, а не `int`.

```
$n = sprintf("%.0f", $f); # округление (а не усечение) до ближайшего целого
```

Чтобы компенсировать смещение, которое всегда вызывает округление дробной части .5 в большую сторону, IEEE требует, чтобы округление выполнялось в направлении ближайшего четного числа. Поэтому следующая инструкция:

```
for (-3 .. 3) { printf "%.0f\n", $_ + 0.5 }
```

выведет последовательность -2, -2, -0, 0, 2, 2 и 4.

ioctl




```
ioctl FILEHANDLE, FUNCTION, SCALAR
```

Реализует системный вызов *ioctl*(2), управляющий вводом-выводом. Чтобы получить правильные определения функций, вначале, вероятно, придется сказать:

```
require "sys/ioctl.ph"; # возможно, /usr/local/lib/perl/sys/ioctl.ph
```

Если *sys/ioctl.ph* не существует или не содержит правильных определений, придется создать собственные, используя существующие заголовочные файлы C, например *sys/ioctl.h*. (Дистрибутив Perl содержит сценарий с именем *h2ph*, который поможет вам это сделать, но запускается он нетривиально.) С аргументом *SCALAR* будет выполнена операция чтения или записи (или обе), в зависимости от *FUNCTION*. Указатель на строковое значение в аргументе *SCALAR* будет передан в третьем аргументе в фактический вызов *ioctl*(2). Если *SCALAR* имеет не строковое, а числовое значение, в вызов будет передано непосредственно это значение, а не указатель на строку. Для работы со значениями в структурах, используемых `ioctl`, хорошо подходят функции `pack` и `unpack`. Если функция `ioctl` должна записать данные в *SCALAR*, необходимо гарантировать, что строка имеет достаточную длину, чего можно добиться, инициализировав строку фиктивными данными нужной длины с помощью `pack`. В следующем примере с помощью `FIONREAD ioctl` определяется, сколько байтов (не символов) доступно для чтения:

```
require "sys/ioctl.ph";

# выделить буфер достаточного размера
$size = pack("L", 0);
ioctl(FH, FIONREAD(), $size)
    || die "Невозможно выполнить ioctl: $!\n";
$size = unpack("L", $size);
```

Ниже показано, как определить текущий размер окна[1] в строках и столбцах:

```
require "sys/ioctl.ph";

# четыре коротких целых без знака (unsigned short)
$template = "S!4";

# выделить буфер достаточного размера
my $ws = pack($template, ());
ioctl(STDOUT, TIOCGWINSZ(), $ws)
    || die "Невозможно выполнить ioctl. $!";
($rows, $cols, $xpix, $ypix) = unpack($template, $ws);
```

Если сценарий *h2ph* не установлен или не работает, можете поискать его с помощью *grep* во включаемых файлах вручную или написать маленькую программу на C для вывода значения. Выяснить формат структуры и требуемый размер для вашей системы можно также, заглянув в программный код на C.

Функции `ioctl` и `fcntl` возвращают значения, перечисленные в табл. 27.2.

---

[1] Точнее, как получить размер окна, связанного с дескриптором файла `STDOUT`.

Таблица 27.2. Значения, возвращаемые функцией ioctl

Значения, возвращаемые системным вызовом	Значения, возвращаемые Perl
-1	`undef`
0	строка `"0 but true"`
все прочее	это же число

Таким образом, Perl возвращает истинное значение при успехе и ложное при неудаче, хотя сохраняется возможность определить фактическое значение, полученное от операционной системы:

```
$retval = ioctl(...) || -1;
printf "ioctl фактически вернула %d\n", $retval;
```

Специальная строка `"0 but true"` избавляет от предупреждений о недопустимых числовых преобразованиях при использовании ключа командной строки *-w* или прагмы `warnings`.

Вызовы `ioctl` нельзя считать переносимыми. Если, скажем, требуется просто отключить эхо-вывод для всего сценария, вот более переносимый способ:

```
system "stty -echo"; # Работает в большинстве систем UNIX.
```

Если вы *можете* делать что-то в Perl, это не значит, что вы *должны* это делать. Лучшую переносимость можно обеспечить с помощью модуля `Term::ReadKey` из CPAN. Практически для любых операций, где могла бы потребоваться функция `ioctl`, в архиве CPAN существуют отдельные модули, реализующие те же операции – и более переносимым способом, поскольку они, как правило, перекладывают тяжелую работу на системный компилятор С.

join

```
join EXPR, LIST
```

Соединяет отдельные строки из списка *LIST* в одну строку, разделяя их значением *EXPR*, и возвращает эту строку. Например:

```
$rec = join ":", $login,$passwd,$uid,$gid,$gcos,$home,$shell;
```

Обратный эффект дает функция `split`. Объединение элементов в поля с фиксированными позициями рассмотрено в описании функции `pack`. Наиболее эффективный способ слияния строк это их объединение (`join`) с пустой строкой в качестве разделителя:

```
$string = join "", @array;
```

В отличие от `split`, `join` не принимает шаблон в первом аргументе. Если вы попытаетесь это сделать, возникнет исключение.

keys




```
keys HASH
keys ARRAY
keys EXPR
```

Возвращает список всех ключей заданного хеша *HASH*. Ключи возвращаются в порядке, кажущемся случайным, но это тот же порядок, который создают функции values и each (в предположении, что хеш не изменялся между вызовами). В качестве побочного эффекта данная функция сбрасывает итератор хеша *HASH*. Вот (довольно примитивный) способ вывести переменные среды:

```
@keys   = keys %ENV;    # ключи следуют в том же порядке, что и
@values = values %ENV; # значения, как видно из вывода этого кода
while (@keys) {
    say pop(@keys), "=", pop(@values);
}
```

Часто желательно при выводе окружения выполнять сортировку по ключам:

```
foreach $key (sort keys %ENV) {
    say $key, "=", $ENV{$key};
}
```

Можно сортировать значения хеша непосредственно, но это довольно бесполезно в отсутствие способа отобразить значения обратно в ключи. Сортировка хеша по значениям обычно реализуется как сортировка по ключам, но при этом используется функция сравнения, которая обращается к значениям, основываясь на ключах. Вот пример числовой сортировки хеша в порядке убывания его значений:

```
foreach $key (sort { $hash{$b} <=> $hash{$a} } keys %hash) {
    printf "%4d %s\n", $hash{$key}, $key;
}
```

Использование keys с хешем, связанным с большим файлом DBM, создаст большой список, а в результате вы получите прожорливый процесс. Более практично в таком случае будет применить функцию each, которая обходит все записи хеша, не создавая при этом гигантского списка.

В скалярном контексте keys возвращает число элементов в хеше (и сбрасывает итератор each). Но чтобы получить эту информацию для связанного хеша, в том числе файла DBM, Perl должен обойти весь хеш, что неэффективно. В таком случае помогает применение keys в пустом контексте.

Применение keys в качестве левостороннего значения увеличит число блоков памяти, выделенных для данного хеша. (Это похоже на предварительное увеличение размера массива путем присваивания большего значения переменной $#array.) Предварительное расширение хеша может повысить эффективность, если известно, что хеш будет расти, и нам заранее известно, как сильно. Если сказать:

```
keys %hash = 1000;
```

для %hash будет выделено не менее 1000 блоков (фактически вы получите 1024 блока, так как округление происходит до следующей степени двойки). Сократить количество блоков, выделенных для хеша, используя keys указанным способом, нельзя (но если – случайно – попытаться сделать это, ничего не произойдет). Блоки сохраняются, даже если выполнить %hash = (). Обратитесь к undef %hash, если требуется освободить память, когда %hash все еще находится в области видимости.

См. также each, values и sort.

kill



```
kill SIGNAL, LIST
```

Посылает сигнал списку процессов. Значением *SIGNAL* может быть целое число или имя сигнала в кавычках (без приставки "SIG"). Попытка указать неизвестное имя сигнала вызовет исключение. Функция возвращает число процессов, которым успешно передан сигнал. Если *SIGNAL* является отрицательным числом, функция передаст сигнал не отдельному процессу, а группе. (В системах UNIX, производных от SysV, отрицательный номер процесса так же вызывает передачу сигнала группе процессов, но такой способ не переносим.) Если PID равен нулю, посылается сигнал всем процессам с таким же идентификатором группы, как у отправителя. Например:

```
$cnt = kill 1, $child1, $child2;
kill 9, @goners;
kill "STOP", getppid  # Можно *так* приостановить свой интерпретатор команд
                      # по умолчанию
        unless getppid == 1; # (Но не дразните init(8).)
```

Сигнал *SIGNAL*, равный 0, проверяет, жив ли процесс и достаточно ли у вас привилегий, чтобы послать ему сигнал. Сигнал при этом не посылается. Таким способом можно удостовериться, что процесс еще существует и не сменил свой UID.

```
use Errno qw(ESRCH EPERM);
if (kill 0 => $minion) {
    say "$minion жив!";
} elsif ($! == EPERM) {                 # изменил UID
    say "$minion вышел из-под контроля!";
} elsif ($! == ESRCH) {
    say "$minion скончался ",           # или стал зомби
} else {
    warn "Странно, не удалось проверить состояние $minion: $!\n"
}
```

См. раздел «Сигналы» главы 15.

last



```
last LABEL
last
```

Оператор `last` осуществляет немедленный выход из данного цикла, как инструкция `break` в C или Java (при использовании в циклах). Если метка *LABEL* отсутствует, оператор относится к самому внутреннему охватывающему циклу. Блок `continue`, если он есть, не выполняется.

```
LINE: while (<MAILMSG>) {
    last LINE if /^$/; # выйти после конца заголовка
    # здесь оставшаяся часть цикла
}
```

`last` нельзя применять для выхода из блока, возвращающего значение, например, `eval {}`, `sub {}` или `do {}`, не должен он использоваться и для выхода из операции `grep` или `map`. Если включен вывод предупреждений, Perl сообщит о применении

last для выхода из цикла, который находится вне текущей лексической области видимости, например, из цикла вызывающей подпрограммы.

Собственно блок семантически идентичен циклу, выполняемому один раз. Поэтому last можно использовать для досрочного выхода из такого блока. В главе 4 приведены примеры, демонстрирующие работу last, next, redo и continue.

lc

```
lc EXPR
lc
```

Возвращает версию *EXPR*, в которой все символы приведены к нижнему регистру. Это – внутренняя функция, реализующая escape-последовательность \L в строках, заключенных в двойные кавычки.

Не используйте lc для реализации сравнения без учета регистра символов, как, возможно, вы делали при работе со строками ASCII, потому что для строк Юникода этот прием дает неверные результаты. В последнем случае лучше использовать функцию fc (выполняющую свертку регистра) из модуля Unicode::CaseFold в CPAN или задействовать прагму use feature "fc" (в Perl версии v5.16 или более поздней). Дополнительные сведения можно найти в разделе «Ошибочные представления о регистре» главы 6.

Функция lc игнорирует коды в диапазоне 128–256, если к строке не применяется семантика Юникода (и не действует режим национальных настроек), о чем трудно догадаться. Особенность unicode_strings гарантирует включение семантики Юникода даже для этих кодов. См. главу 6.

Текущее значение LC_CTYPE учитывается, если действует use locale, хотя взаимодействие национальных настроек с Юникодом является, как говорится, предметом текущих исследований. Последние результаты можно найти на страницах руководства *perllocale* (*http://perldoc.perl.org/perllocale.html*), *perlunicode* (*http://perldoc.perl.org/perlunicode.html*) и *perlfunc* (*http://perldoc.perl.org/perlfunc.html*).

lcfirst

```
lcfirst EXPR
lcfirst
```

Возвращает вариант строки *EXPR* с первым символом в нижнем регистре. Это – внутренняя функция, реализующая escape-последовательность \l в строках, заключенных в двойные кавычки. См. предыдущую статью (lc), где приводятся дополнительные предупреждения, касающиеся регистра символов в Юникоде.

length



```
length EXPR
length
```

Возвращает длину скалярной величины *EXPR* в кодах (символах, видимых программисту). При вызове без аргумента возвращает длину $_. (Однако смотрите, чтобы дальнейший код сценария не выглядел как начало *EXPR*, иначе лексический анализатор Perl собьется. Например, length < 10 не будет компилироваться. Если сомневаетесь, используйте круглые скобки.)

Не пытайтесь применять length для определения размера массива или хеша. Для массива следует использовать scalar @array, а для хеша – scalar keys %hash. (вызов scalar обычно опускается, если является избыточным.)

Чтобы найти длину строки в графемах (символах, видимых пользователю), их можно просто подсчитать:

```
my $count = 0;
$count++ while our $string =~ /\X/g;
```

или воспользоваться модулем Unicode::GCString из CPAN, позволяющим работать со строками, как с последовательностями графем, а не кодов. Этот модуль позволяет также определить длину строки в позициях вывода. При таком подходе сохраняется возможность использовать printf для форматирования вывода, а если включить творческую жилку, вы сможете даже использовать format и write, несмотря на то, что код при выводе может занимать одну или две позиции или вообще не занимать позиций.

__LINE__

Специальная лексема, компилирующаяся в номер текущей строки. См. раздел «Генерирование Perl в других языках» в главе 21.

link

```
link OLDFILE, NEWFILE
```

Создает новое имя файла, являющееся ссылкой на старое имя файла. Возвращает истинное значение в случае успеха и ложное в противном случае. См. также symlink далее в этой главе. Вероятно, что эта функция окажется реализованной только в UNIX-системах.

listen



```
listen SOCKET, QUEUESIZE
```

Предписывает системе принимать подключения через сокет *SOCKET* и помещать запросы на подключение, ожидающие обработки, в очередь размером *QUEUESIZE*. Представьте себе, что на телефоне есть очередь ожидающих вызовов, в которую могут встать до 17 позвонивших. (Мурашки по коже!) Функция возвращает истинное значение в случае успеха и ложное в противном случае.

```
use Socket;
listen(PROTOSOCK, SOMAXCONN)
    || die "невозможно установить очередь ожидания на PROTOSOCK: $!"
```

См. описание функции accept. См. также раздел «Сокеты» главы 15 и описание *listen*(2).

local

```
local EXPR
```

Этот оператор не создает локальную переменную, для этой цели применяется my. Вместо этого он локализует существующие переменные, т.е. в результате одна

или более переменных получают значения с локальной областью видимости в самом внутреннем охватывающем блоке, eval или файле. Если перечислено несколько переменных, список следует заключить в круглые скобки, поскольку этот оператор связывает сильнее, чем запятые. Все переменные должны быть разрешенными левосторонними конструкциями, т.е. допускать присваивание значений. В их число могут входить отдельные элементы массивов и хешей.

При выполнении этого оператора текущие значения указанных переменных сохраняются в скрытом стеке и восстанавливаются после выхода из блока, подпрограммы, eval или файла. После выполнения local, но перед выходом из области видимости, все подпрограммы и выполняемые форматы будут видеть локальное (внутреннее) значение, а не предыдущее (внешнее), поскольку переменная остается глобальной, несмотря на наличие у нее локального значения. На техническом языке это называется *динамической областью видимости*. См. раздел «Объявления с областью видимости» главы 4.

Выражению *EXPR* можно при желании присваивать значение, что позволяет инициализировать переменные при их локализации. В отсутствие инициализатора все переменные получают значение undef, а все массивы и хеши – значение (). Как и при обычном присваивании, если переменные заключены в круглые скобки (или если переменная является массивом или хешем), выражение справа вычисляется в списочном контексте. В противном случае выражение справа вычисляется в скалярном контексте.

В любом случае выражение справа вычисляется перед локализацией, но инициализация происходит после локализации, поэтому локализованную переменную можно инициализировать ее нелокализованным значением. К примеру, следующий код показывает, как выполнить временную модификацию глобального массива:

```
if ($sw eq "-v") {
    # инициализировать локальный массив глобальным
    local @ARGV = @ARGV;
    unshift(@ARGV, "echo");
    system @ARGV;
}
# @ARGV восстановлен
```

Можно также временно модифицировать глобальные хеши:

```
# временно добавить пару записей к хешу %digits
if ($base12) {
    # (ЗАМЕЧАНИЕ: Мы не утверждаем, что это эффективно!)
    local(%digits) = (%digits, T => 10, E => 11);
    parse_num();
}
```

Посредством local можно присвоить временные значения отдельным элементам массивов и хешей, даже имеющим лексическую область видимости:

```
if ($protected) {
    local $SIG{INT} = "IGNORE";
    precious(); # никаких прерываний во время выполнения этой функции
}               # предыдущий обработчик (если он был) восстановлен
```

Можно также использовать local с переменными типа typeglob для создания локальных дескрипторов файлов без загрузки объемистых модулей объектов:

```
local *MOTD;                # защита глобальных дескрипторов MOTD
my $fh = do { local *FH }; # создать новый косвенный дескриптор
```

В старом коде еще можно увидеть локализованные переменные typeglob, необходимые для создания новых дескрипторов файлов, однако в настоящее время достаточно простого my $fh, **поскольку, если передать в качестве аргумента неопределенную переменную туда, где ожидается реальный дескриптор файла (например, в первом аргументе функции** open **или** socket**) Perl «автоматически оживит» для вас новенький дескриптор файла.**

В общем случае следует использовать my, **поскольку** local **означает совсем не то, что большинство понимает как «локальный». См. описание** my.

Для локального удаления элементов хеша или массива, т.е. для удаления в текущем блоке, можно использовать конструкцию delete local *EXPR*.

localtime

```
localtime EXPR
localtime
```

Преобразует значение, возвращаемое time, **в список из девяти элементов, представляющих время, приведенное к местному часовому поясу. Обычно она используется так:**

```
# 0    1    2     3     4    5     6     7     8
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime;
```

Если, как в данном случае, аргумент *EXPR* **опущен, выполняется** localtime(time()).

Все элементы списка представляют собой числа и поступают прямо из struct tm. **(Это жаргон программирования на С – пусть он вас не беспокоит.) В частности, это означает, что** $mon **имеет диапазон** 0..11, **и январю соответствует число** 0, **а** $wday **имеет диапазон** 0..6, **и воскресенье обозначается числом** 0. **Несложно запомнить, какие величины имеют нумерацию с нуля, поскольку именно они всегда служат индексами массивов, содержащих названия месяцев и дней недели и также имеющих нумерацию с нуля.**

Например, получить название текущего дня недели можно так:

```
$thisday = ("Sun","Mon","Tue","Wed","Thu","Fri","Sat")[(localtime)[6]];
```

$year **– количество лет, прошедших после 1900; т.е. в 2023 году** $year **будет иметь значение** 123, **а не просто** 23. **Чтобы получить четырехзначный год, просто скажите** $year + 1900. **Чтобы получить двузначный год (например, «01» в 2001 году), используйте** sprintf("%02d", $year % 100).

Модуль библиотеки Perl Time::Local **содержит подпрограмму** timelocal, **осуществляющую преобразование в обратном направлении.**

В скалярном контексте localtime **возвращает строку в стиле** ***ctime*(3)**. **Например, команду** ***date*(1)** **можно (почти полностью)[1] эмулировать с помощью:**

---

[1] ***date*(1)** **выводит часовой пояс, а** localtime **– нет.**

```
perl -le 'print scalar localtime()'
```

См. также описание функции strftime **в стандартном модуле** POSIX**; она реализует более развитые возможности форматирования времени. Модуль** Time::localtime **поддерживает интерфейс к этой функции через имена.**

lock

```
lock THING
```

Устанавливает блокировку на переменную, подпрограмму или объект, на который ссылается *THING*, **пока блокировка не выйдет из области видимости. Для обратной совместимости эта функция реализуется как встроенная, но только если версия Perl скомпилирована с поддержкой потоков выполнения и только в области действия прагмы** use Threads. **В противном случае Perl воспринимает ее как функцию, определенную пользователем.**

log



```
log EXPR
log
```

Возвращает натуральный логарифм (т.е. по основанию *e*) *EXPR*. **Для отрицательных значений** *EXPR* **возбуждается исключение. Чтобы получить логарифм по другому основанию, используйте элементарную алгебру: логарифм числа по основанию *N* равен натуральному логарифму этого числа, деленному на натуральный логарифм *N*. Например:**

```
sub log10 {
    my $n = shift;
    return log($n)/log(10);
}
```

Обратной для log **функцией является** exp. **См.** exp.

lstat



```
lstat EXPR
lstat
```

Действует подобно функции stat **(включая установку специального дескриптора файла _), но если файл является символической ссылкой, выполняется** stat **для собственно символической ссылки, а не для файла, на который она указывает. (Если символические ссылки не реализованы в системе, выполняется обычный вызов** stat.**)**



m//



```
/PATTERN/
m/PATTERN/
```

Выполняет поиск по шаблону, где *PATTERN* **служит регулярным выражением. В программе этот оператор интерпретируется как строка в двойных кавычках, а не функция. См. главу 5.**

map

```
map BLOCK LIST
map EXPR, LIST
```

Вычисляет *BLOCK* или *EXPR* для каждого элемента списка *LIST* (поочередно присваивая локализованной переменной $_ значение каждого элемента) и возвращает список, объединяющий результаты всех вычислений. Блок *BLOCK* или выражение *EXPR* вычисляется в списочном контексте, поэтому каждый элемент *LIST* может отображаться в ноль, в один или несколько элементов возвращаемого значения. Все они помещаются в один в один плоский список. Например:

```
@words = map { split ' ' } @lines;
```

превратит список строк в список слов. Но часто существует взаимно однозначное соответствие между исходными и конечными значениями:

```
@chars = map chr, @nums;
```

превратит список чисел в список соответствующих символов. А вот пример отображения один к двум:

```
%hash = map { genkey($_) => $_ } @array;
```

что является лишь необычным вариантом следующей конструкции:

```
%hash = ();
for my $_ (@array) {
    $hash{genkey($_)} = $_;
}
```

Поскольку $_ является псевдонимом (неявной ссылкой) значения списка, эту переменную можно использовать для изменения элементов массива. Это удобно и поддерживается, хотя может порождать странные результаты, когда *LIST* не является именованным массивом. Применение для этих целей обычного цикла `foreach` может оказаться более прозрачным. См. также `grep`. Функция `map` отличается от `grep` тем, что возвращает список результатов всех последовательных вычислений *EXPR*, тогда как `grep` возвращает список всех значений *LIST*, для которых *EXPR* имеет истинное значение.

mkdir



```
mkdir FILENAME, MASK
mkdir FILENAME
```

Создает каталог с именем *FILENAME* и правами доступа, определяемыми числовой маской *MASK* с учетом текущего значения `umask`. В случае успеха возвращает истинное значение, в противном случае – ложное.

Если аргумент *MASK* опущен, используется маска 0777, что почти всегда соответствует желаемому значению. В целом предпочтительнее создавать каталоги с маской *MASK*, дающей более широкие полномочия (например, 0777), и позволять пользователю модифицировать ее с помощью собственного значения `umask`, чем задавать ограничивающую маску, отбирая у пользователя возможность сужать эти полномочия. Исключение составляют каталоги и файлы, которые должны быть закрытыми (например, почтовые файлы). См. `umask`.

Если системный вызов *mkdir*(2) не встроен в библиотеку C, Perl эмулирует его, вызывая программу *mkdir*(1) для каждого каталога. При необходимости создания большого списка каталогов в такой системе эффективнее вручную вызвать программу *mkdir* со списком каталогов, чем запускать тьму подпроцессов.

msgctl

$! X U

```
msgctl ID, CMD, ARG
```

Выполняет системный вызов System V IPC *msgctl*(2); подробности см. в *msgctl*(2). Может потребоваться сначала выполнить `use IPC::SysV`, чтобы получить определения констант. Если в *CMD* передается значение `IPC_STAT`, то в *ARG* должна передаваться переменная, куда будет помещаться возвращаемая структура C `msqid_ds`. Возвращаемые значения – такие же, как в `ioctl` и `fcntl`: `undef` означает ошибку, `"0 but true"` означает ноль, и фактическое возвращаемое значение в остальных случаях.

`msgctl` доступна только в системах, поддерживающих механизмы System V IPC, а таких значительно меньше, чем поддерживающих сокеты.

msgget

$! X U

```
msgget KEY, FLAGS
```

Выполняет системный вызов System V IPC *msgget*(2). Подробности см. в *msgget*(2). Возвращает идентификатор очереди сообщений или `undef` в случае ошибки. Перед вызовом необходимо выполнить `use IPC::SysV`.

Доступна только в системах, поддерживающих механизмы System V IPC.

msgrcv




```
msgrcv ID, VAR, SIZE, TYPE, FLAGS
```

Выполняет системный вызов *msgrcv*(2), чтобы извлечь сообщение из очереди сообщений *ID* в переменную *VAR*, ограничив размер сообщения значением *SIZE*. Подробности см. в *msgrcv*(2). После получения сообщения в начале *VAR* будет находиться тип сообщения, а максимальная длина *VAR* вычисляется как *SIZE* плюс размер типа сообщения. Декодировать содержимое переменной можно вызовом `unpack("l! a*")`. Функция возвращает истинное значение в случае успеха и ложное, если возникла ошибка. Перед вызовом необходимо выполнить `use IPC::SysV`.

Доступна только в системах, поддерживающих механизмы System V IPC.

msgsnd




```
msgsnd ID, MSG, FLAGS
```

Выполняет системный вызов *msgsnd*(2) для отправки сообщения *MSG* в очередь сообщений *ID*. Подробности см. в *msgsnd*(2). Аргумент *MSG* должен начинаться с длинного целого, представляющего тип сообщения. Создать сообщение можно так:

```
$msg = pack "l! a*", $type, $text_of_message;
```

Возвращает истинное значение в случае успеха и ложное, если возникла ошибка. Перед вызовом необходимо выполнить `use IPC::SysV`.

Доступна только в системах, поддерживающих механизмы System V IPC.

my

```
my TYPE EXPR : ATTRIBUTES
my EXPR   ATTRIBUTES
my TYPE EXPR
my EXPR
```

Этот оператор объявляет одну или более закрытых переменных, которые будут существовать только в самом внутреннем охватывающем блоке, подпрограмме, eval или файле. Если перечислено несколько переменных, список необходимо заключить в круглые скобки, поскольку этот оператор связывает сильнее, чем запятые. Объявлять таким способом можно только простые скаляры или целые массивы и хеши.

Имя переменной не может квалифицироваться именем пакета, поскольку все переменные пакета глобально доступны через соответствующие им таблицы символов, а лексические переменные не связаны с какими-либо таблицами символов. Поэтому, в отличие от local, этот оператор не имеет к глобальным переменным никакого отношения, за исключением того, что маскирует все переменные с такими же именами в своей области видимости (т.е. там, где существуют закрытые переменные). Однако доступ к глобальной переменной всегда можно получить, квалифицировав ее именем пакета, или через символическую ссылку.

Область видимости закрытой переменной начинается лишь командой, находящейся после объявления такой переменной. Область видимости переменной простирается на все последующие вложенные блоки, вплоть до конца области видимости самой переменной.

Однако это означает, что любые подпрограммы, которые вызываются из области видимости закрытой переменной, не могут видеть эту закрытую переменную, если только блок с определением подпрограммы не вложен текстуально в область видимости этой переменной. Это звучит сложно, пока вы с этим не разберетесь. На техническом языке это называется *лексической областью видимости* (*lexical scoping*), поэтому такие переменные мы часто называем *лексическими*. В культуре C их часто называют «автоматическими» переменными, поскольку они автоматически создаются и удаляются при входе и выходе из области видимости.

Выражению *EXPR* можно при желании присваивать значение, что позволяет инициализировать лексические переменные. (Если инициализатор отсутствует, все скаляры получают неопределенное значение, а все массивы и хеши – пустой список). Как и при обычном присваивании, если переменные слева заключаются в круглые скобки (или если переменная является массивом или хешем), выражение справа вычисляется в списочном контексте. В противном случае выражение справа вычисляется в скалярном контексте. Например, формальным параметрам подпрограммы можно дать имена с помощью списочного присваивания:

```
my ($friends, $romans, $countrymen) = @_;
```

Следите за тем, чтобы не забыть о скобках, указывающих на списочный контекст присваивания, например:

```
my $country = @_; # правильно или нет?
```

Здесь переменной присваивается длина массива (т.е. число аргументов подпрограммы), поскольку массив вычисляется в скалярном контексте. Однако скаляр-

ное присваивание формальному параметру возможно посредством оператора shift. На практике, поскольку методам объектов в первом аргументе передается объект, многие подпрограммы методов начинают с того, что «крадут» первый аргумент:

```
sub simple_as {
    my $self = shift;    # присваивание скаляра
    my ($a,$b,$c) = @_; # присваивание списка
    ...
}
```

Если попытаться объявить подпрограмму с лексической областью видимости как my sub, Perl завершит работу сообщением, что эта функция пока не реализована. (И, конечно, Perl не завершится, если эта функция *уже* реализована.[1])

Параметры *TYPE* и *ATTRIBUTES* являются необязательными. Вот как выглядит объявление с использованием этих параметров:

```
my Dog $spot :ears(short) :tail(long);
```

TYPE, если задан, определяет тип скаляра или скаляров, объявленных в *EXPR*, либо непосредственно, как одна или более скалярных переменных, либо косвенно, через массив или хеш. Если *TYPE* является именем класса, предполагается, что скаляры содержат ссылки на объекты этого типа или объекты, совместимые с этим типом. В частности, совместимыми считаются производные классы. Следовательно, если предположить, что Collie происходит от Dog, можно объявить:

```
my Dog $lassie = new Collie;
```

Данное объявление сообщает, что объект $lassie будет использоваться подобно объекту Dog. То обстоятельство, что фактически это объект Collie, не должно иметь значения до тех пор, пока с ним делают то, что можно делать с Dog. Благодаря магии виртуальных методов, в классе Collie вполне может присутствовать собственная реализация этих методов Dog, но объявление выше говорит только об интерфейсе, а не о реализации. Теоретически.

В действительности, на сегодняшний день Perl не придает большого значения информации о типе, но она доступна для усовершенствований в будущем. (Традиционно эта информация использовалась в псевдохешах, но они уже сошли со сцены.) Объявление *TYPE* следует рассматривать как родовой интерфейс, который когда-нибудь будет реализовываться различными способами в зависимости от класса. На самом деле, *TYPE* может даже не являться официальным именем класса. Мы зарезервировали имена типов в нижнем регистре для Perl, потому что один из способов, которыми мы хотели бы расширить интерфейс типа, состоит в разрешении необязательных объявлений типов низкого уровня, таких как int, num и str.[2] Эти объявления не будут использоваться для контроля типов; они будут служить советами компилятору в отношении того, как оптимизировать хранение переменной, предполагая, что переменная по преимуществу будет использоваться так,

---

1 На это есть определенные надежды, так как опыт разработки Perl 6 демонстрирует, что подпрограммы могут быть по умолчанию лексическими, и при этом использовать их будет так же просто.

2 Фактически именно такой синтаксис поддержки встроенных типов в настоящее время исследуется в Perl 6, поэтому парни из Perl 5 смогут позаимствовать все самое лучшее, когда парни из Perl 6 решат все проблемы. :–)

как объявлена. Семантика скаляров останется, в основном, той же – по-прежнему можно будет складывать два скаляра str или выводить скаляр int, как если бы они были обычными полиморфными скалярами, с которыми вы уже знакомы. Но для переменной, объявленной как int, Perl мог бы хранить только целое значение и забыть о кэшировании конечной строки, которое выполняется сейчас. Ускорились бы циклы с управляющей переменной типа int, особенно в коде, скомпилированном в C. В частности, числовые массивы можно было бы хранить гораздо более компактным способом. В конечном итоге, встроенная функция vec может выйти из употребления, если мы разрешим писать такие объявления:

```
my bit @bitstring;
```

Объявление *ATTRIBUTES* используется чаще, чем определения типов; подробнее об этом читайте в описании прагмы attributes в главе 29. Первым атрибутом, который мы когда-нибудь реализуем, будет, вероятно, constant:

```
my num $PI : constant = atan2(1,1) * 4;
```

Но есть и другие идеи, такие как определение значений по умолчанию для массивов и хешей или разрешение совместного использования переменных несколькими взаимодействующими интерпретаторами. Подобно интерфейсу типа, интерфейс атрибутов должен рассматриваться как родовой интерфейс, своеобразное рабочее место для изобретения нового синтаксиса и семантики. Мы не знаем, как будет развиваться Perl в следующие десять лет. Мы знаем только, что можем облегчить себе жизнь, планируя заранее это развитие.

См. также local, our и раздел «Объявления с областью видимости» главы 4.

new

```
new CLASSNAME LIST
new CLASSNAME
```

В Perl отсутствует встроенная функция new. Это просто обычный метод-конструктор (т.е. подпрограмма, определяемая пользователем), которая определяется или наследуется классом *CLASSNAME* (т.е. пакетом), чтобы дать возможность создавать объекты типа *CLASSNAME*. Многие конструкторы именуются «new», но лишь по договоренности, призванной обмануть программистов на C++, чтобы им казалось, что они понимают происходящее. Всегда читайте документацию класса, чтобы узнать, как вызывать его конструкторы; например, конструктор, создающий Tk-виджет списка, называется просто Listbox. См. главу 12.

next




```
next LABEL
next
```

Оператор next действует подобно команде continue в языке C: он начинает очередную итерацию цикла, обозначенного меткой *LABEL*:

```
LINE: while (<STDIN>) {
    next LINE if /^#/; # пропустить комментарии
    ...
}
```

Если бы в этом примере присутствовал блок continue, он был бы выполнен сразу после вызова next. Если метка *LABEL* опущена, оператор ссылается на самый внутренний охватывающий цикл.

Собственно блок семантически идентичен циклу, выполняемому один раз. Поэтому next осуществляет досрочный выход из такого блока (через блок continue, если он есть).

Нельзя организовывать при помощи next выход из блока, который возвращает значение, например eval {}, sub {} или do {}, кроме того, этот оператор не должен применяться для выхода из операции grep или map. Если включен вывод предупреждений, Perl сообщит, когда next используется для выхода из цикла, который находится вне текущей лексической области видимости, например, цикла в вызывающей подпрограмме. См. раздел «Операторы циклов» в главе 4.

no



```
no MODULE VERSION LIST
no MODULE VERSION
no MODULE LIST
no MODULE
no VERSION
```

Ознакомьтесь с описанием оператора use, который представляет собой своего рода противоположность no. Большинство стандартных модулей не осуществляет обратного импорта чего-либо, что, в сущности, делает no пустой операцией. Модули прагм склонны быть здесь более любезными. Если *MODULE* не удается найти, возникает исключение.

oct



```
oct EXPR
oct
```

Интерпретирует *EXPR* как восьмеричную строку и возвращает эквивалентное десятичное значение. Если строка *EXPR* начинается с "0x", она интерпретируется как шестнадцатеричное число, а если с "0b" – как двоичное. Следующий код правильно преобразует любые строковые представления чисел в десятичной, двоичной, восьмеричной и шестнадцатеричной форме записи, стандартной для Perl, в соответствующие целые числа:

```
$val = oct $val if $val =~ /^0/;
```

Обратное преобразование можно выполнить посредством sprint, используя подходящий формат:

```
$dec_perms = (stat("filename"))[2] & 07777;
$oct_perm_str = sprintf "%o", $dec_perms;
```

Функция oct обычно используется, когда необходимо преобразовать строку данных, такую как "644", например, в маску прав доступа к файлу. Хотя Perl автоматически преобразует строки в числа по мере надобности, такое автоматическое преобразование предполагает использование основания 10.

Ведущие пробелы и любые нецифровые символы в конце, такие как десятичная точка, игнорируются без вывода каких-либо предупреждений (oct обрабатывает

только неотрицательные целые числа и не может обрабатывать отрицательные числа или числа с плавающей запятой).

open

$! X ARG X 1 X U

```
open FILEHANDLE, MODE, EXPR, LIST
open FILEHANDLE, MODE, EXPR
open FILEHANDLE, MODE, REFERENCE
open FILEHANDLE, EXPR
```

Функция open связывает внутренний дескриптор файла *FILEHANDLE* со спецификацией внешнего файла, определяемой аргументами *EXPR* или *LIST*. Эту функцию можно вызывать с одним, двумя или тремя аргументами (или более, если третий аргумент является командой). Если аргументов три или более аргументов, второй аргумент *MODE* указывает режим доступа, в котором должен быть открыт файл, а третий аргумент – фактическое имя файла или выполняемую команду, в зависимости от режима. Вместе с командой можно передать дополнительные аргументы, если потребуется вызвать команду, например, system или exec, непосредственно, без вовлечения интерпретатора команд. Команду можно также передать единым аргументом (третьим), и в таком случае решение о вызове командного интерпретатора зависит от наличия в этом аргументе метасимволов. (Не используйте более трех аргументов, если аргументы представляют собой обычные имена файлов: это не сработает.) Если режим *MODE* не распознан, open возбуждает исключение.

Особый случай представляет версия с тремя аргументами, определяющая режим для чтения/записи, когда в третьем аргументе передается undef:

```
open(my $tmp, "+>", undef) or die ...
```

Этот вызов откроет дескриптор файла для безымянного временного файла. Допускается также симметричный режим "+<", но, прежде чем что-то прочесть из временного файла, это «что-то» нужно сначала записать в него. Перед чтением придется воспользоваться функцией seek.

Версию open с тремя аргументами можно использовать, чтобы включить фильтры ввода/вывода (иногда они называются «дисциплинами»), реализующие промежуточную обработку входных и выходных данных (см. описание модуля PerlIO). Например:

```
open(my $fh, "< :encoding(UTF-8)", "filename")
    || die "can't open UTF-8 encoded filename: $!";
```

откроет файл в кодировке UTF-8 (т.е. файл, содержащий символы Юникода). Начиная с версии v5.14, в случае ошибки декодирования входных потоков в кодировке UTF-8, исключение *не* возбуждается. Если вы используете какой-либо фильтр поддержки UTF-8, рассмотрите возможность добавления директивы:

```
use warnings FATAL => "utf8";
```

чтобы иметь возможность перехватывать исключения. См. главу 6.

Обратите внимание, что при использовании фильтров в open с тремя аргументами, фильтры по умолчанию, хранящиеся в ${^OPEN}, игнорируются. (См. главу 25; фильтры по умолчанию устанавливаются прагмой open или ключом *-CioD*.)

Если ваша версия Perl была собрана с использованием PerlIO[1], дескрипторы файлов можно открывать непосредственно в скаляры Perl, передавая ссылки на эти скаляры в аргументе *EXPR* вызова open с тремя аргументами:

```
open($fh, ">", \$variable) || ..
```

Чтобы повторно открыть дескриптор файла STDOUT или STDERR для вывода в файл, хранящийся в памяти, дескриптор сначала нужно закрыть:

```
close(STDOUT)                  || die "Невозможно закрыть STDOUT: $!";
open(STDOUT, ">", \$variable) || die "Невозможно открыть STDOUT в памяти: $!";
```

Если аргументов только два, предполагается, что режим и имя файла/команда объединены во втором аргументе. (Если во втором аргументе указать только имя файла, по соображениям безопасности файл будет открыт только для чтения.)

```
open(LOG, "> logfile")    or die "Невозможно создать logfile: $!"; # хорошо
open(LOG, ">", "logfile") or die "Невозможно создать logfile: $!"; # еще лучше
```

Функция open возвращает истинное значение в случае успеха и undef – в противном случае. Если open открывает канал в дочерний процесс, она вернет идентификатор этого процесса. Как и для любого системного вызова, всегда проверяйте значение, возвращаемое open, чтобы убедиться в успехе операции[2]. Но это не C или Java, поэтому избегайте команды if, если можно ограничиться оператором ||. Допускается также использовать or, и в этом случае можно опустить ( ) в вызове open. Если вы решили опустить скобки и превратить функцию в оператор списка, следите, чтобы за списком следовала конструкция or die, а не || die, потому что приоритет || выше, чем у списочных операторов, и || будет относиться к последнему аргументу, а не к результату open в целом:

```
open LOG, ">", "logfile" || die "Невозможно создать logfile: $!"; # НЕВЕРНО
open LOG, ">", "logfile" or die "Невозможно создать logfile: $!"; # ok
```

В такой форме записи трудно разобраться, поэтому, чтобы было видно, где заканчивается оператор списка, обычно вводят скобки:

```
open(LOG, ">", "logfile") or die "Невозможно создать logfile: $!" # хорошо
open(LOG, ">", "logfile") || die "Невозможно создать logfile: $!" # хорошо
```

или просто переносят or на другую строку:

```
open LOG, ">", "logfile"
    or die "Невозможно создать logfile: $!";
```

Как видно из примера, аргумент *FILEHANDLE* часто является простым идентификатором (обычно в верхнем регистре), но может также быть выражением, возвращающим ссылку на дескриптор файла. (Это может быть символическая ссылка на имя дескриптора файла или жесткая ссылка на любой объект, который может интерпретироваться как дескриптор файла.) Это называется *косвенным дескриптором файла* (*indirect filehandle*), и любая функция, принимающая *FILEHANDLE* в первом аргументе, может обрабатывать косвенные дескрипторы файлов так же, как прямые. Но open имеет одну особенность: если передать ей в качестве косвенного

---

1 Используется по умолчанию, начиная с версии v5.8, выпущенной в 2002

2 Если не используете прагму autodie, которая позаботится о проверке за вас.

дескриптора файла неопределенную переменную, Perl автоматически определит эту переменную, т. е. «самооживит» ее, поместив в нее ссылку на надлежащий дескриптор файла. Одно из вытекающих преимуществ состоит в том, что дескриптор файла будет автоматически закрыт, когда не останется действующих ссылок на него, например, когда переменная выйдет из области видимости:

```
{
    my $fh;                    # (не инициализирована)
    open($fh, ">", "logfile") # $fh самооживлена
        or die "Невозможно создать logfile: $!";
                               # сделать что-то с $fh
}                              # здесь $fh закроется
```

Объявление my $fh можно хорошо читаемым образом включить в open:

```
open(my $fh, ">", "logfile") || die ...
```

Символ > перед именем файла является примером режима. Изначально существовала форма open с двумя аргументами. Недавнее добавление формы с тремя аргументами позволяет отделить режим от имени файла, чтобы они не смешивались. В следующем примере предполагается, что пользователь не пытается открыть файл, имя которого начинается символом ">". Мы можем быть уверены, что открывается в режиме ">" (записи) файл с именем в *EXPR*, при этом файл создается, если не существует, и размер файла усекается до нуля, если файл уже существует:

```
open(LOG, ">", "logfile") || die "Невозможно создать logfile: $!";
```

В более коротких формах имя файла и режим находятся в одной строке. Строка анализируется во многом так же, как анализируется перенаправление файлов и каналов интерпретатором команд системы. Во-первых, из строки удаляются ведущие и замыкающие пробельные символы. Затем строка исследуется с обоих концов, если нужно, на наличие символов, указывающих, как должен быть открыт файл. Между режимом и именем файла допускается наличие пробельного символа.

Режимы открытия файлов являются символами перенаправления в стиле интерпретатора команд. Список этих символов приведен в табл. 27.3. (Если требуется открыть файл в смешанном режиме доступа, не входящем в данную таблицу, обращайтесь к функции низкого уровня sysopen.)

Таблица 27.3. Режимы для open

Режим	Доступ для чтения	Доступ для записи	Только для дополнения	Создать несуществующий	Разрушить существующий
< *PATH*	Да	Нет	Нет	Нет	Нет
> *PATH*	Нет	Да	Нет	Да	Да
>> *PATH*	Нет	Да	Да	Да	Нет
+< *PATH*	Да	Да	Нет	Нет	Нет
+> *PATH*	Да	Да	Нет	Да	Да
+>> *PATH*	Да	Да	Да	Да	Нет
| *COMMAND*	Нет	Да	неприменимо	неприменимо	неприменимо
COMMAND |	Да	Нет	неприменимо	неприменимо	неприменимо

Если установлен режим "<" или режим вообще не установлен, существующий файл открывается для чтения. В режиме ">" файл открывается для записи, что приводит к усечению существующих файлов и созданию несуществующих, а в режиме ">>" файл создается при необходимости и открывается на запись в режиме, когда все данные автоматически записываются в конец файла. Если новый файл создается при использовании режима ">" или ">>", а файл ранее не существовал, права доступа определяются текущим значением umask процесса согласно правилам, описанным для этой функции.

Вот типичные примеры:

```
open(INFO,      "datafile") || die("невозможно открыть datafile: $!");
open(INFO,    "< datafile") || die("невозможно открыть datafile: $!");
open(RESULTS, "> runstats") || die("невозможно открыть runstats: $!");
open(LOG,     ">> logfile ") || die("невозможно открыть logfile:  $!");
```

Если вы предпочитаете сокращать количество знаков пунктуации, можете написать так:

```
open INFO,      "datafile" or die "невозможно открыть datafile: $!";
open INFO,    "< datafile" or die "невозможно открыть datafile: $!";
open RESULTS, "> runstats" or die "невозможно открыть runstats: $!";
open LOG,     ">> logfile " or die "невозможно открыть logfile:  $!";
```

При открытии для чтения специальное имя файла "-" обозначает STDIN. При открытии для записи это же специальное имя файла обозначает STDOUT. Обычно они задаются как "<-" и ">-" соответственно.

```
open(INPUT,  "-" ) || die; # повторно открыть стандартный ввод для чтения
open(INPUT,  "<-") || die; # то же, но явно
open(OUTPUT, ">-") || die; # повторно открыть стандартный вывод для записи
```

Благодаря этому программе можно передать имя файла, которое вынудит использовать стандартный ввод или вывод, но автору программы не придется писать код, чтобы узнать это.

Любой из этих трех режимов можно также предварить знаком «+», чтобы открыть файл одновременно для чтения и записи. Тем не менее, знаки «меньше» и «больше» по-прежнему определяют, будет ли файл усечен или создан, а также должен ли он уже существовать. Это означает, что "+<" почти всегда лучше для целей чтения/обновления, а сомнительный режим "+>" разрушит содержимое файла, прежде чем можно будет что-нибудь из него прочесть. (Устанавливайте этот режим, только если необходимо заново прочесть то, что было только что записано.)

```
open(DBASE, "+< database")
    || die "невозможно открыть существующую БД в режиме обновления: $!";
```

Файл, открытый в режиме обновления, можно рассматривать как базу данных с произвольным доступом и перемещаться к байту с заданным номером при помощи seek, но переменная длина записей в обычных текстовых файлах обычно делает непрактичным использование режима чтения/обновления таких файлов. Иной подходе к обновлению предлагается в описании параметра командной строки *-i* в главе 17.

Если ведущий символ в *EXPR* является символом канала (конвейера), open создаст новый процесс и соединит с ним дескриптор файла, открытый только для записи.

В результате можно осуществлять запись по этому дескриптору, и записанное будет попадать на стандартный ввод команды. Например:

```
open(PRINTER, "| lpr -Plp1")      || die "невозможно выполнить: $!";
print PRINTER "stuff\n";
close(PRINTER)                    || die "сбой lpr/close: $?/$!";
```

Если замыкающий символ в *EXPR* является символом канала, open также запустит новый процесс, но на этот раз подключит к нему дескриптор файла, доступный только для чтения. Благодаря этому вывод команды в STDOUT появится в вашем дескрипторе для чтения. Например:

```
open(NET, "netstat -i -n |")      || die "невозможно выполнить: $!";
while (<NET>) { ... }
close(NET)                        || die "невозможно закрыть netstat: $!/$?";
```

Явно закрывая дескриптор файла канала, родительский процесс будет ожидать, когда порожденный процесс завершится и вернет код состояния в $? ($CHILD_ERROR). Возможно также, что close установит $! ($OS_ERROR). Примеры интерпретации кодов ошибок вы найдете в описании close и system.

Любая команда в конвейере, содержащая метасимволы интерпретатора команд, например, маски или символы перенаправления ввода/вывода, передается каноническому системному интерпретатору команд (*/bin/sh* в UNIX), чтобы сначала были обработаны эти специфические для интерпретатора команд конструкции. Если метасимволы не обнаружены, Perl запустит новый процесс сам, не вызывая оболочку.

Для запуска конвейера можно использовать версию open с тремя аргументами. Эквивалентом предшествующего открытия канала в таком стиле будет:

```
open(PRINTER, "|-", "lpr -Plp1")    || die "невозможно выполнить: $!";
open(NET, "-|", "netstat -i -n")    || die "невозможно выполнить: $!";
```

Здесь знак «минус» во втором аргументе представляет команду в третьем аргументе. В данном примере интерпретатор команд не вызывается, но если необходимо гарантировать отсутствие обработки интерпретатором, в новых версиях Perl можно сказать:

```
open(PRINTER, "|-", "lpr", "-Plp1")     || die "невозможно выполнить: $!";
open(NET, "-|", "netstat", "-i", "-n")  || die "невозможно выполнить: $!";
```

Если для открытия канала в специальную команду или из специальной команды "-"[1] используется версия open с двумя аргументами, сначала неявно вызывается fork. (В системах, где ветвление не поддерживается, возникает исключение. В системах Microsoft fork не поддерживалась до конца двадцатого столетия, но теперь поддерживается.) В данном случае знак «минус» представляет новый порожденный процесс, являющийся копией родительского. Возвращаемым значением вызова open, выполнившего ветвление, является идентификатор дочернего процесса в родительском процессе, 0 – в порожденном процессе, и значение undef при неудаче fork (в этом случае порожденный процесс не запускается). Например:

---

[1] Можно представлять себе это как пропуск команды в вышеприведенных версиях с тремя аргументами.

```
my $pid = open(FROM_CHILD, "-|") // die "невозможно выполнить : $!";

if ($pid) {
    @parent_lines = <FROM_CHILD>, # код родителя
}
else {
    print STDOUT @child_lines;    # код потомка
}
```

В родительском процессе дескриптор файла ведет себя как обычно, но в порожденном процессе STDOUT (или STDIN) оказывается связан с вводом (или выводом) родительского процесса. Порожденный процесс не видит открытого дескриптора файла родительского процесса. (На это указывает PID, равный 0.)

В общем случае эту конструкцию следует использовать вместо обычного вызова open, открывающего канал, если требуется больший контроль над выполнением команды в конвейере (например, при выполнении setuid), и нет желания проверять команды на присутствие метасимволов. Следующие операции открытия каналов примерно эквивалентны:

```
open(FH,          "| tr   'a-z'  'A-Z'");          # канал в команду интерпретатора
open(FH, "|-",        "tr", "a-z", "A-Z" );        # канал в голую команду
open(FH, "|-") || exec("tr", "a-z", "A-Z") || die; # канал в порожденный процесс
open(FOO, "|-",       "tr"  "a-z", "A-Z") || die;  # канал в порожденный процесс
```

как и эти:

```
open(FH,              "cat -n 'file' |");           # канал из команды интерпретатора
open(FH, "-|",        "cat", "-n", "file");         # канал из голой команды
open(FH, "-|") || exec "cat", '-n", "file") || die; # канал из порожденного процесса
open(FOO, "-|", "cat", "-n", $file)         || die; # канал из порожденного процесса
```

Последние два примера в каждом блоке демонстрируют использование канала в «форме списка», которая поддерживается пока не на всех платформах. Основное правило гласит, что если платформа поддерживает истинную функцию fork (другими словами, если платформа – UNIX), можно использовать форму списка.

Дополнительные примеры по этой теме можно найти в разделе «Анонимные каналы» главы 15. Более интересные примеры применения open, порождающей процессы, вы найдете в разделах «Разговор с самим собой» главы 15 и «Наведение порядка в окружающей среде» главы 20.

Прежде чем выполнить ветвление процессов, Perl старается вытолкнуть буферы для всех файлов, открытых для записи, но на некоторых платформах эта особенность может быть не реализована. Для большей безопасности может потребоваться установить переменную $| ($AUTOFLUSH в модуле English) или вызвать метод autoflush модуля IO::Handle для всех открытых дескрипторов файлов.

В системах, поддерживающих флаг закрытия при exec, он будет устанавливаться для вновь открываемых дескрипторов файлов, в соответствии со значением $^F ($SYSTEM_FD_MAX).

Попытка закрыть любой канал приостанавливает родительский процесс до завершения порожденного процесса и затем возвращает код состояния в $? и ${^CHILD_ERROR_NATIVE}.

В именах файлов, передаваемых open с двумя аргументами, любые начальные и конечные пробельные символы удаляются, а обычные символы перенаправления сохраняются. Эта особенность известна как «волшебное открытие» и часто используется с выгодой. Пользователь получает возможность указать имя файла вроде "rsh cat file |", а вы – возможность изменять некоторые имена файлов по своему усмотрению:

```
$filename =~ s/(.*\.gz)\s*$/gzip -dc < $1|/;
open(FH, $filename) || die "Невозможно открыть $filename: $!";
```

При запуске команды с помощью open необходимо выбрать ввод либо вывод: "cmd|" для чтения либо "|cmd" для записи. Нельзя с помощью open запустить команду и открыть канал одновременно на ввод и на вывод, что, казалось бы, позволяет недопустимое (в данное время) обозначение "|cmd|". Однако стандартные библиотечные подпрограммы IPC::Open2 и IPC::Open3 дают близкий эквивалент. Подробности о двунаправленных каналах читайте в разделе «Двунаправленная связь» главы 15.

В традициях оболочки Борна можно также начать выражение *EXPR* сочетанием >&; в таком случае оставшаяся часть строки интерпретируется как имя дескриптора файла (или указатель файла, если это число), который должен быть дублирован посредством системного вызова *dup2*(2).[1] Символ & можно использовать после >, >>, <, +>, +>> и +<. (Режим должен соответствовать режиму исходного дескриптора файла.)

Это может понадобиться, если существует открытый дескриптор файла, и есть желание создать еще один дескриптор, являющийся точной копией первого.

```
open(SAVEOUT, ">&SAVEERR") || die "невозможно дублировать SAVEERR: $!"
open(MHCONTEXT, "<&4")     || die "невозможно дублировать fd4: $!";
```

Это значит, что если функция рассчитывает получить имя файла, но вы не хотите давать ей имя файла, потому что уже открыли файл, можете просто передать ей дескриптор файла, предварив его амперсандом. Однако лучше всего использовать полностью квалифицированный дескриптор файла, поскольку функция может находиться в другом пакете:

```
somefunction("&main::LOGFILE");
```

Другой причиной дублировать дескриптор файла может быть желание временно переадресовать текущий дескриптор, не потеряв при этом первоначального адресата. Следующий сценарий запоминает, переадресовывает и восстанавливает STDOUT и STDERR:

```
#!/usr/bin/perl
use v5.14;
open SAVEOUT, ">&STDOUT";
open SAVEERR, ">&STDERR";

open(STDOUT, ">foo.out") || die "невозможно перенаправить stdout";
open(STDERR, ">&STDOUT") || die "невозможно дублировать stdout";
```

---

[1] Это (пока) не работает для объектов ввода/вывода, созданных путем самооживления дескриптора файла, но для получения дескриптора файла и его дублирования всегда можно использовать fileno.

```
select STDERR; $| = 1         # включить автоматическую очистку буфера
select STDOUT; $| = 1         # включить автоматическую очистку буфера

say STDOUT "stdout 1";        # эти потоки ввода/вывода распространяются
say STDERR "stderr 1";        # также в подпроцессы

system("some command");       # использует новые stdout/stderr

close STDOUT;
close STDERR;

open STDOUT, ">&SAVEOUT";
open STDERR, ">&SAVEERR";

say STDOUT "stdout 2";
say STDERR "stderr 2";
```

Если дескриптору файла или номеру его указателя предшествует комбинация &=, а не просто &, вместо создания совершенно нового дескриптора файла Perl сделает *FILEHANDLE* псевдонимом существующего дескриптора с помощью функции *fdopen*(3) из библиотеки C. При этом несколько экономятся системные ресурсы, хотя в теперешние времена это не столь важно.

```
$fd = $ENV{"MHCONTEXTFD"};
open(MHCONTEXT, "<&=$fdnum")
    || die "невозможно выполнить fdopen для дескриптора $fdnum: $!";
```

Дескрипторы файлов STDIN, STDOUT и STDERR всегда остаются открытыми при выполнении exec. Другие дескрипторы файлов, по умолчанию, – нет. В системах, поддерживающих функцию fcntl, можно модифицировать флаг дескриптора файла для закрытия при выполнении exec.

```
use Fcntl qw(F_GETFD F_SETFD);
$flags = fcntl(FH, F_SETFD, 0)
        || die "Невозможно сбросить флаг закрытия при exec для FH: $!"
```

См. также описание специальной переменной $^F ($SYSTEM_FD_MAX) в главе 25.

Используя open с двумя аргументами, следует проявлять осторожность, задействуя строковую переменную в качестве имени файла, поскольку она может содержать произвольные таинственные символы (особенно если имя файла передано произвольными таинственными личностями через Интернет). Если не проявить осторожность, часть имени файла может быть интерпретирована как строка *MODE*, пробельный символ, который можно опустить, признак необходимости дублирования или знак «минус». Вот один исторически интересный способ принять меры:

```
$path =~ s#^(\s)#./$1#;
open (FH, "< $path\0")     || die "невозможно открыть $path: $!";
```

И даже такой способ имеет несколько слабых мест. Лучше обратитесь к версии open с тремя аргументами, чтобы открывать произвольное имя файла аккуратно и без (дополнительной) угрозы для безопасности:

```
open(FH, "<", $path) || die "невозможно открыть $path: $!";
```

С другой стороны, если вам действительно нужен системный вызов *open*(2) в стиле C со всеми сопутствующими деталями и украшениями, возьмите sysopen:

```
use Fcntl;
sysopen(FH, $path, O_RDONLY) || die "невозможно открыть $path: $!"
```

Если система различает текстовые и двоичные файлы, может потребоваться установить двоичный режим (или воздержаться от этого), чтобы избежать искажения файлов. В таких системах результат попытки открыть двоичный файл в текстовом режиме или текстовый файл в двоичном режиме может вам не понравиться.

Системы, где без binmode не обойтись, отличаются от остальных по формату, используемому для текстовых файлов. В системах, где binmode не требуется, каждая строка завершается одним символом, соответствующим тому, что C понимает как перевод строки, – \n. В эту категорию попадает UNIX, включая современные версии Mac OS. VMS, MVS, MS-всякие и прочие операционные системы типа S&M реализуют ввод/вывод текстовых и двоичных файлов по-разному, поэтому в них нужно использовать binmode.

Или ее эквивалент. Двоичный режим можно устанавливать в функции open, не вызывая binmode отдельно. В аргументе *MODE* (но только в версии с тремя аргументами) можно подключать различные фильтры ввода и вывода. Чтобы обойтись без binmode, используйте версию open с тремя аргументами и упомяните фильтр :raw после остальных символов *MODE*:

```
open(FH, "<:raw", $path) || die "невозможно открыть $path: $!";
```

См. описание модуля Encode в главе 6, где описывается, что еще можно помещать сюда, включая обработку текстовых файлов в Windows.

opendir




```
opendir DIRHANDLE, EXPR
```

Открывает каталог с именем *EXPR* для работы с ним с помощью функций readdir, telldir, seekdir, rewinddir и closedir. Возвращает истинное значение в случае успеха. Дескрипторам каталогов отводится собственное пространство имен, отделенное от пространства имен дескрипторов файлов.

ord

$_

```
ord EXPR
ord
```

Возвращает числовое значение (код) первого символа в *EXPR*. Возвращаемое значение всегда беззнаковое. Если нужно получить значение со знаком, используйте unpack("c", EXPR). Чтобы преобразовать все символы строки в список чисел, выполните unpack("U*", EXPR). Чтобы отыскать код символа по его строковому имени, используйте функцию charnames::vianame из прагмы charnames.

our

```
our TYPE EXPR : ATTRIBUTES
our EXPR : ATTRIBUTES
our TYPE EXPR
our EXPR
```

Оператор our объявляет одну или несколько переменных как глобальные в охватывающем блоке, eval или файле. Объявление our подчиняется таким же прави-

лам видимости, как объявление my, но не создает новую закрытую переменную, а просто разрешает свободное обращение к существующей в пакете глобальной переменной. Если перечислено несколько переменных, список должно заключить в круглые скобки.

Основная цель объявления our – скрыть переменную от действия объявления use strict "vars"; поскольку переменная маскируется под переменную my, можно использовать объявленную таким образом глобальную переменную, не квалифицируя ее именем пакета. Однако, как и в случае переменной my, это правило действует только в пределах лексической области видимости объявления our. В этом отношении оно отличается от директивы use vars, воздействующей на весь пакет и не ограничивающейся лексической областью видимости.

Объявление our похоже на my и в том, что позволяет объявлять переменные с параметрами *TYPE* и *ATTRIBUTES*. Например:

```
our Dog $spot :ears(short) :tail(long);
```

На момент написания этих строк не вполне ясно, во что это выльется. Атрибуты могли бы воздействовать на глобальную или локальную интерпретацию $spot. С одной стороны, было бы более всего похоже на переменные my, если бы атрибуты охватывали текущее локальное представление $spot, не трогая другие представления глобальной переменной в иных местах. С другой стороны, если в одном модуле объявлено, что $spot – это Dog, а в другом, что $spot – это Cat, можно прийти к мяукающим собакам и лающим котам. Это составляет предмет проводимых в настоящее время исследований, – это мы так витиевато признаемся, что пока не понимаем того, что обсуждаем.

Другое сходство между our и my заключается в области видимости. Объявление our объявляет глобальную переменную, которая будет видима во всей лексической области видимости, даже через границы пакетов. Пакет, которому будет принадлежать переменная, определяется местонахождением объявления, а не точкой использования. Это значит, что следующее поведение можно считать характерной особенностью:

```
package Foo;
our $bar;      # $bar представляет собой $Foo::bar до конца лексической области
$bar = 582;

package Bar;
print $bar;    # выводит 582, как если бы "our" было "my"
```

Одно из отличий состоит в том, что my создает новую закрытую переменную, а our делает видимой существующую глобальную переменную, что особенно важно в операциях присваивания. Если объединить присваивание на этапе выполнения с объявлением our, значение, присвоенное глобальной переменной, не исчезнет, когда our выйдет из области видимости. Для этого потребуется local:

```
($x, $y) = ("один", "два");
say "перед блоком x равно $x, y равно $y";
{
    our $x = 10;
    local our $y = 20;
    say "в блоке x равно $x, y равно $y";
```

```
}
say "после блока x равно $x, y равно $y";
```

Вывод будет таким:

```
перед блоком x равно один, y равно два
в блоке x равно 10, y равно 20
после блока x равно 10, y равно два
```

Многократные объявления our в одной и той же лексической области допускаются, если они находятся в разных пакетах. Если они находятся в одном пакете, Perl выдает предупреждения, когда его об этом просят.

```
use warnings;
package Foo;
our $bar;       # объявляет $Foo::bar до конца лексической области
$bar = 20;

package Bar;
our $bar = 30; # объявляет $Bar::bar до конца лексической области
print $bar;     # выведет 30

our $bar;       # выведет предупреждение
```

См. описание local, my и state, а также раздел «Объявления с областью видимости» главы 4.

pack



```
pack TEMPLATE, LIST
```

Принимает список *LIST* обычных значений Perl, преобразует их в строку байтов согласно шаблону *TEMPLATE* и возвращает эту строку. Шаблоны для функций pack и unpack описываются в главе 26.

package

```
package NAMESPACE VERSION BLOCK
package NAMESPACE VERSION
package NAMESPACE BLOCK
package NAMESPACE
```

На самом деле, это не функция, а объявление, которое говорит, что блок *BLOCK* или оставшаяся часть наиболее глубоко вложенной охватывающей области видимости принадлежит указанной таблице символов или пространству имен. (Область видимости объявления package, таким образом, такая же, как у объявлений my, state или our.) Внутри своей области видимости объявление заставляет компилятор разрешать все неквалифицированные глобальные идентификаторы путем поиска их в таблице символов объявленного пакета.

Объявление package действует только на глобальные переменные – в том числе те, с которыми использован оператор local, – не лексические переменные, созданные с помощью my, state или our. Оно воздействует только на неквалифицированные глобальные переменные – глобальные переменные, квалифицированные именем собственного пакета, игнорируют текущий объявленный пакет. Глобальные переменные, объявленные с помощью our, – неквалифицированные и потому учи-

тывают текущий пакет, но только в точке объявления, после чего ведут себя как переменные my. То есть в оставшейся части лексической области видимости переменные our «прикрепляются» к пакету, используемому в точке объявления, даже если будет встречено новое объявление пакета.

Обычно объявление package помещается в начало файла, который должен быть включен с помощью оператора require или use, но можно поместить его в любое место, где допускается инструкция. При создании файла обычного или объектно-ориентированного модуля пакету часто дают то же имя, что и файлу, чтобы избежать путаницы. (Принято также именовать такие пакеты с заглавной буквы, поскольку модули, названия которых начинаются со строчной буквы, принято интерпретировать как модули прагм.)

Можно переключаться в конкретный пакет более чем в одном месте. Это влияет только на выбор таблицы символов, используемой компилятором в оставшейся части блока. (Если компилятор видит другое объявление package на том же уровне, новое объявление получает более высокий приоритет, чем предыдущее.) Предполагается, что основная программа начинается с невидимого объявления package main.

Если присутствует аргумент *VERSION*, оператор package присваивает переменной $VERSION, находящейся в данном пространстве имен, объект version с указанным значением *VERSION*. Аргумент *VERSION* должен содержать номер версии в «строгом» формате, как определено прагмой version: положительное десятичное число (целое или вещественное) без обозначения экспоненты или v-строка в десятично-точечной нотации с ведущим символом v и, как минимум, с тремя компонентами. (В будущем это требование может быть ослаблено до двух компонентов, по мере того, как программисты начнут привыкать пользоваться объектами версий.) Переменная $VERSION должна определяться в пакете только один раз.

Ссылаться на переменные, подпрограммы, указатели и форматы в других пакетах можно, квалифицируя идентификатор именем пакета и двойным двоеточием: $Package::Variable. Если имя пакета не указано, предполагается пакет main. Это значит, что $::sail эквивалентно $main::sail, а также $main'sail, что иногда еще можно встретить в старом коде.

Следующий пример:

```
package main;        $sail = "hale and hearty";
package Mizzen;      $sail = "tattered";
package Whatever;
say "My main sail is $main::sail.";
say "My mizzen sail is $Mizzen::sail."
```

Выведет:

```
My main sail is hale and hearty.
My mizzen sail is tattered.
```

Таблица символов пакета хранится в хеше, имя которого заканчивается двумя двоеточиями. К примеру, таблица символов пакета main называется %main::. Поэтому к символу *main::sail можно также обратиться конструкцией $main::{"sail"}.

Дополнительные сведения о пакетах вы найдете в главе 10. Описание my (приведенное ранее в этой главе) дает информацию по другим вопросам, связанным с областями видимости.

__PACKAGE__

Специальная лексема, возвращающая имя пакета, в котором она встретилась. См. главу 10.

pipe

```
pipe READHANDLE, WRITEHANDLE
```

Как и соответствующий системный вызов, открывает пару связанных каналов – см. *pipe*(2). Обычно вызывается непосредственно перед fork, после которого программа чтения из канала должна закрыть *WRITEHANDLE*, а программа записи закрыть *READHANDLE*. (Иначе канал скроет EOF от программы чтения, когда будет закрыт программой записи.) Когда два процесса соединяются каналами, обеспечивающими двунаправленный обмен, может произойти взаимоблокировка, если не проявить особую осторожность. Кроме того, обратите внимание, что каналы Perl используют буферизацию стандартного ввода/вывода, поэтому может потребоваться установить $| ($OUTPUT_AUTOFLUSH) для *WRITEHANDLE*, чтобы очищать буфер после каждой операции вывода в зависимости от приложения – см. select (дескриптор выходного файла).

(Как и в open, если один из файловых дескрипторов не определен, он будет «самооживлен».)

Вот небольшой пример:

```
pipe(README, WRITEME);
unless ($pid = fork) {                          # потомок
    defined($pid) || die "невозможно создать потомка: $!";
    close(README);
    for $i (1..5) { print WRITEME "строка $i\n" }
    exit;
}
$SIG{CHLD} = sub { waitpid($pid, 0) }
close(WRITEME);
@strings = <README>;
close(README);
print "Получено:\n", @strings;
```

Обратите внимание, как записывающий код закрывает конец для чтения, а читающий код закрывает конец для записи. Нельзя организовать двустороннюю связь при помощи одного канала. Для этого необходимы два разных канала или системный вызов socketpair. См. раздел «Каналы» главы 15.

pop



```
pop ARRAY
pop
```

Данная функция интерпретирует массив *ARRAY* как стек – она выталкивает (удаляет) и возвращает последнее значение в массиве, укорачивая массив на один элемент. Если *ARRAY* опущен, выталкивает @_ в лексической области видимости подпрограмм и форматов; выталкивает @ARGV в области видимости файлов (обычно основной программы) или внутри лексических областей видимости, установленных

конструкциями eval *STRING*, BEGIN {}, CHECK {}, UNITCHECK {} и END {}. Оказывает такой же эффект, как:

```
$tmp = $ARRAY[$#ARRAY--];
```

или:

```
$tmp = splice @ARRAY, -1;
```

Если в массиве нет элементов, возвращает undef. (Но не полагайтесь на это, чтобы определять момент, когда массив опустеет, если он содержит значения undef!) См. также push и shift. Если требуется вытолкнуть более одного элемента, используйте splice.

Функция pop требует, чтобы ее первый аргумент был массивом, а не списком. Если требуется просто извлечь последний элемент списка, используйте

```
( LIST )[-1]
```

Начиная с версии v5.14, функция pop может принимать ссылку на «неосвященный» массив, которая будет разыменована автоматически. Данная особенность pop пока является экспериментальной. Ее поведение может измениться в будущих версиях Perl.

pos



```
pos SCALAR
pos
```

Возвращает позицию в *SCALAR*, где закончился последний поиск m//g в этом скаляре.

Возвращает смещение символа (кода), *следующего за* последним найденным символом (равносильно выражению length($`) + length($&)). С этой позиции начнется следующий поиск m//g в данной строке. Запомните, что смещение начала строки равно 0. Обратите внимание, что значение 0 является допустимым смещением, полученным в результате поиска. Значение undef указывает, что позиция поиска сброшена (обычно из-за отсутствия соответствия, но это же значение возвращается, если поиск в строке вообще не производился.)

Например:

```
$graffito = "fee fie foe foo";
    while ($graffito =~ m/e/g) {
        say pos $graffito;
}
```

выведет 2, 3, 7 и 11, т.е. смещения всех символов, следующих за "e". Функции pos можно присвоить значение, чтобы сообщить, где нужно начать следующий поиск m//g:

```
$graffito = "fee fie foe foo";
pos $graffito = 4; # Пропустить fee, начать с fie
while ($graffito =~ m/e/g) {
    say pos $graffito;
}
```

Этот фрагмент выведет только 7 и 11. Утверждение \G в регулярном выражении соответствует только текущей позиции, которая указывается функцией pos для строки, где производится поиск. См. раздел «Позиции» главы 5.

Обратите внимание, что здесь говорится о кодах символов, а не о символах. Нам не хотелось бы вводить вас в заблуждение. Коды – это символы, видимые программисту, некоторые из них могут быть невидимы для пользователя. Символы, которые видит пользователь, и которые обычно называются графемами или групповыми графемами, могут состоять из нескольких кодов. Например, последовательность "\r\n" – это один символ для пользователя, но два символа для программиста. Если вам необходима версия pos, работающая с графемами, а не с кодами, загляните в модуль Unicode::GCString, опубликованный в CPAN.

print



```
print FILEHANDLE LIST
print FILEHANDLE
print LIST
print
```

Выводит строку или список строк, разделенных запятыми. Если установлена переменная $\ ($OUTPUT_RECORD_SEPARATOR), ее содержимое выводится в конце списка. Функция возвращает истинное значение в случае успеха и ложное в противном случае. Между элементами списка *LIST* выводится текущее значение переменной $, (если установлено). В конце списка *LIST* выводится текущее значение переменной $\ (если установлено).

Чтобы вывести содержимое $_ в *FILEHANDLE*, следует использовать действительный дескриптор файла, такой как FH, а не косвенный, как $fh. *FILEHANDLE* может быть именем скалярной переменной (без индекса), и тогда переменная содержит либо имя фактического дескриптора файла, либо ссылку на какой-либо объект дескриптора файла. Как и любой другой косвенный объект, *FILEHANDLE* может также быть блоком, возвращающим уместное значение:

```
print { $OK ? "STDOUT" : "STDERR" } "всячина\n";
print { $iohandle[$i] } "всячина\n";
```

Если аргумент *FILEHANDLE* является переменной, а очередная лексема – термом, он может быть ошибочно принят за оператор, если не вставить знак + или не заключить аргументы в скобки, например:

```
print $a - 2;  # выведет $a - 2 в дескриптор по умолчанию (обычно STDOUT)
print $a (- 2);# выведет -2 в дескриптор, заданный в $a
print $a -2;   # выведет -2 (таинственные правила синтаксического анализа :-)
```

Если *FILEHANDLE* опущен, осуществляется вывод в дескриптор файла, выбранный в настоящий момент; первоначально это STDOUT. Чтобы по умолчанию для вывода использовался другой дескриптор, отличный от STDOUT, используйте операцию select *FILEHANDLE*.[1] Если *LIST* тоже опущен, выводится содержимое $_. Поскольку print принимает список, содержимое *LIST* вычисляется в списочном контексте. Поэтому, если сказать:

```
print OUT <STDIN>;
```

---

[1] Поэтому в действительности STDOUT не является дескриптором файла по умолчанию для print. Это лишь *значение по умолчанию* для дескриптора файла по умолчанию.

будет выводиться не очередная строка, а все оставшиеся строки из стандартного ввода вплоть до конца файла, потому что именно это <STDIN> **возвращает в списочном контексте. Если требуется другое, скажите:**

```
print OUT scalar <STDIN>;
```

Кроме того, помня правило «все, что похоже на функцию, функцией и является», следите за тем, чтобы за ключевым словом print **не следовала левая круглая скобка, если только вы не хотите, чтобы соответствующая правая круглая скобка завершала аргументы** print, **– вставьте знак + или заключите в круглые скобки все аргументы:**

```
print (1+2)*3, "\n";    # НЕВЕРНО
print +(1+2)*3, "\n";   # ok
print ((1+2)*3, "\n");  # ok
```

При вызове с аргументом *FILEHANDLE* **аргумент** *LIST* **можно опустить, только если** *FILEHANDLE* **является обычным «голым словом» дескриптора файла, а не блоком или косвенным дескриптором файла.**

```
$_ = "всячина\n";
*NEWOUT = *STDOUT;
print NEWOUT;           # ok: выведет "всячина\n"

$fh = *NEWOUT;
print $fh;              # НЕВЕРНО: выведет STDOUT "*main::STDOUT"
```

Вывод Юникода в дескриптор файла, при открытии которого не был упомянут фильтр ввода/вывода, определяющий порядок кодирования, приводит к неотключаемому предупреждению «Wide character in print» (широкие символы в выводе). Чтобы исправить проблему, укажите кодировку с помощью binmode **или передайте ее во втором аргументе версии функции** open **с тремя аргументами.**

```
binmode(STDOUT, ":utf8") || die "Ошибка вызова binmode: $!";
open(HANDLE, "> :encoding(UTF-16)", $file)
    || die "Невозможно открыть файл $file: $!";
```

Вывод в закрытый канал или сокет вызовет появление сигнала SIGPIPE. **См. раздел «Сигналы» в главе 15.**

printf



```
printf FILEHANDLE FORMAT, LIST
printf FORMAT, LIST
printf FILEHANDLE
printf LIST
printf
```

Выводит форматированную строку в *FILEHANDLE* **или, если он опущен, в текущий дескриптор файла; первоначально это** STDOUT. **Первым элементом** *LIST* **должна быть строка, в которой указано, как форматировать остальные элементы. Действует аналогично функциям** ***printf*(3)** **и** ***fprintf*(3)** **из библиотеки С и эквивалентна следующей конструкции:**

```
print FILEHANDLE sprintf FORMAT, LIST
```

за исключением того, что $\ ($OUTPUT_RECORD_SEPARATOR) не добавляется в конец вывода.

Описание порядка интерпретации форматов приводится в главе 26. Мы могли бы продублировать всю приведенную в этой главе информацию здесь, но эта книга и так уже стала экологической катастрофой.

Исключение возникает, только если в аргументе *FILEHANDLE* задан недопустимый тип ссылки.

Если опустить оба аргумента, *FORMAT* и *LIST*, используется $_ – но в таком случае следовало остановить выбор на print. Не попадайтесь в ловушку, применяя printf там, где достаточно обычной print. Функция print более эффективна и менее подвержена ошибкам.

prototype



```
prototype FUNCTION
```

Возвращает прототип функции в виде строки (или undef, если у функции нет прототипа). *FUNCTION* представляет собой ссылку на функцию или имя функции, прототип которой требуется получить.

Если *FUNCTION* является строкой, начинающейся последовательностью CORE::, оставшаяся часть считается именем встроенной функции Perl, и если такой встроенной функции нет, возникает исключение. Если встроенная функция не *переопределяема* (например, qw//) или ее аргументы нельзя выразить прототипом (например, system), функция возвращает undef, поскольку встроенная функция ведет себя в действительности не так, как функция Perl. В противном случае возвращается строка, описывающая эквивалентный прототип.

push



```
push ARRAY, LIST
```

Интерпретирует массив *ARRAY* как стек и вталкивает значения из списка *LIST* в конец *ARRAY*. Длина *ARRAY* возрастает на длину *LIST*. Возвращает новую длину массива. Функция push действует так же, как:

```
for my $value (listfunc()) {
    $array[++$#array] = $value;
}
```

или:

```
splice @array, @array, 0, listfunc();
```

но более эффективно (для вас и вашего компьютера). Функцию push можно сочетать с функцией shift, чтобы создать быстродействующий регистр сдвига или очередь:

```
for (;;) {
    push @array, shift @array;
    ...
}
```

См. также pop и unshift.

Начиная с версии v5.14, push может принимать ссылку на «неосвященный» хеш или массив, которая будет разыменована автоматически. Эта особенность push считается экспериментальной. Ее поведение может измениться в будущем.

q/STRING/

```
q/STRING/
qq/STRING/
qr/STRING/
qw/STRING/
qx/STRING/
```

Обобщенные кавычки. См. раздел «Выберите собственные кавычки» главы 2. Сведения о коде завершения qx// вы найдете в описании readpipe, а сведения о коде завершения qr// – в описании m//. См. также раздел «Управление процессом» главы 5.

quotemeta

$_

```
quotemeta EXPR
quotemeta
```

Возвращает значение *EXPR*, добавляя обратную косую черту перед всеми символами, не являющимися буквами или цифрами. (То есть, обратная косая черта появится перед всеми символами, не входящими в множество /[A-Za-z_0-9]/, независимо от национальных настроек.) Это – внутренняя функция, реализующая escape-последовательность \Q в интерполируемых контекстах (строках в двойных кавычках, обратных апострофах и шаблонах).

rand

```
rand EXPR
rand
```

Возвращает псевдослучайное число с плавающей запятой, большее или равное 0 и меньшее значения *EXPR*. (Значение *EXPR* должно быть положительным.) Если выражение *EXPR* опущено, возвращает число с плавающей запятой между 0 и 1 (включая 0, но исключая 1). rand автоматически вызывает функцию srand, если та еще не вызывалась. См. также srand.

Чтобы получить случайное целое число, например, для моделирования игры в кости, объедините rand с int, как показано ниже:

```
$roll = int(rand 6) + 1; # $roll теперь число между 1 и 6
```

Поскольку Perl использует функцию псевдослучайных чисел из библиотеки C, например *random*(3) или *drand48*(3), качественное распределение не гарантируется. Если требуется более сильная случайность, например, для целей криптографии, можете обратиться к документации по *random* (4) (если у вас в системе есть устройство */dev/random* или */dev/urandom*), к модулям Math::Random::Secure, Math::Random::MT::Perl и Math::TrulyRandom в CPAN или прочитать хороший учебник по генерации псевдослучайных чисел на компьютерах, например второй том Кнута.[1]

---

[1] Д. Кнут «Искусство программирования. Получисленные алгоритмы», том 2, 3-е издание. – Пер. с англ. – Вильямс, 2011.

read




```
read FILEHANDLE, SCALAR, LENGTH, OFFSET
read FILEHANDLE, SCALAR, LENGTH
```

Пытается прочесть *LENGTH* ***символов*** (в смысле кодов символов, а не графем) в переменную *SCALAR* из дескриптора файла *FILEHANDLE*. Возвращает число прочитанных символов или 0, если достигнут конец файла. В случае ошибки возвращает `undef`. *SCALAR* растет или сокращается в зависимости от фактически прочитанной длины. Значение *OFFSET*, если задано, определяет смещение в переменной *SCALAR*, благодаря чему запись можно выполнять в середину строки.

Чтобы скопировать данные из дескриптора файла *FROM* в дескриптор файла *TO*, можно сказать:

```
while (read(FROM, $buf, 16384)) {
    print TO $buf;
}
```

Обратите внимание на слово «*символов*»: в зависимости от параметров дескриптора файла, читаться будут либо байты (восьмибитные значения), либо символы. Байты – это всего лишь способ представления недекодированных кодов символов с маленькими значениями в Perl. По умолчанию все дескрипторы файлов оперируют байтами. Но если дескриптор файла открыт, например, с фильтром ввода/вывода `:utf8`, операции будут выполняться над символами Юникода в кодировке UTF-8, а не над байтами. Аналогичный фильтр ввода/вывода можно упомянуть при вызове `binmode` с двумя аргументами, в среднем аргументе `open` или в прагме `open`: во всех этих случаях можно будет прочитать практические любые символы.

Противоположность `read` составляет простая функция `print`, которая уже знает длину записываемой строки и может записать строку любой длины. Было бы ошибкой использовать `write`, работающую исключительно с форматами.

Функция `read` реализована через функцию стандартного ввода/вывода *fread*(3), поэтому фактический системный вызов *read*(2) может прочесть больше символов, чем определено аргументом *LENGTH*, чтобы заполнить входной буфер, а *fread*(3) может выполнить более одного системного вызова *read*(2), чтобы заполнить буфер. Чтобы получить более полный контроль, используйте `syread`, указывая требуемый системный вызов. Вызовы `read` и `sysread` не должны перемежаться (разве что вам требуется сильнодействующее колдовство или головная боль).

readdir




```
readdir DIRHANDLE
```

Читает записи каталога (являющиеся простыми именами файлов) из дескриптора каталога, открытого функцией `opendir`. В скалярном контексте возвращает очередную запись, если таковая имеется, в противном случае возвращает `undef`. В списочном контексте возвращает все оставшиеся записи или пустой список, если записей больше нет. Например:

```
opendir(THISDIR, "  ") || die "serious dainbramage: $!",
@allfiles = readdir THISDIR;
closedir THISDIR,
say "@allfiles";
```

Этот код выведет имена всех файлов в текущем каталоге в одну строку. Если требуется исключить записи "." и "..", используйте любое из следующих заклинаний, которое сочтете наиболее понятным:

```
@allfiles = grep { $_ ne "." && $_ ne ".." } readdir THISDIR;
@allfiles = grep { ! /^[.][.]?\z/ } readdir THISDIR;
@allfiles = grep { ! /^\.{1,2}\z/ } readdir THISDIR;
@allfiles = grep !/^\.\.?\z/, readdir THISDIR;
```

Чтобы избежать вывода всех файлов .* (как в программе *ls*):

```
@allfiles = grep !/^\./, readdir THISDIR;
```

Чтобы получить только текстовые файлы:

```
@textfiles = grep -T, readdir THISDIR;
```

Но будьте внимательны с последним вариантом, потому что к результату readdir нужно снова «приклеить» часть, указывающую каталог, если это не текущий каталог, например:

```
opendir(THATDIR, $path) || die "невозможно выполнить opendir $path: $!";
@dotfiles = grep { /^\./ && -f } map { "$path/$_" } readdir(THATDIR);
closedir THATDIR;
```

Начиная с версии v5.12 вызов readdir можно использовать в цикле while. В этом случае очередной результат будет сохраняться в переменной $_. В качестве дескриптора можно также использовать скалярную переменную с неопределенным значением. В этом случае она автоматически будет инициализирована анонимным дескриптором каталога.

```
my $dh;                   # гарантировать создание новой переменной
opendir($dh, $somedir) || die "невозможно opendir $somedir: $!";
while (readdir($dh)) {
    print "$somedir/$_\n";
}
closedir $dh;
```

readline




```
readline FILEHANDLE
readline
```

Реализующая оператор <*FILEHANDLE*> внутренняя функция, с которой можно работать непосредственно. Функция читает очередную запись из *FILEHANDLE*, который может быть именем дескриптором файла или выражением косвенного дескриптора файла, возвращающим либо имя фактического дескриптора файла, либо ссылку на нечто напоминающее объект дескриптора файла, например typeglob. В скалярном контексте каждый вызов считывает и возвращает очередную запись, пока не будет достигнут конец файла, и тогда очередной вызов вернет undef. В списочном контексте readline читает записи, пока не будет достигнут конец файла, и возвращает список записей. Под «записью» обычно понимается строка текста, но если присвоить переменной $/ ($INPUT_RECORD_SEPARATOR) иное значение, отличное от значения по умолчанию, этот оператор начнет «нарезать» текст по-другому. Неопределенное значение в $/ увеличит размер фрагмента до размеров целого файла.

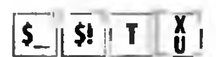
При чтении файлов целиком в скалярном контексте, когда файл пуст, первый вызов readline вернет "", а каждый следующий – undef. При чтении файлов целиком из волшебного дескриптора файла ARGV, для каждого файла возвращается один фрагмент (и здесь пустые файлы возвращаются как ""), а когда все файлы будут прочитаны, последует одно значение undef. Если *FILEHANDLE* опущен, используется дескриптор файла ARGV.

Оператор *<FILEHANDLE>* более подробно описывается в разделе «Операторы ввода» главы 2.

```
$line = <STDIN>;
$line = readline(STDIN);   # то же самое
$line = readline(*STDIN);  # то же самое
$line = readline(\*STDIN); # то же самое

open(my $fh, "<&=STDIN") || die;
bless($fh => "AnyOldClass");
$line = readline($fh);     # то же самое
```

readlink



```
readlink EXPR
readlink
```

Возвращает имя файла, на который указывает символическая ссылка. Выражение *EXPR* должно возвращать путь к файлу, последний компонент которого представляет символическую ссылку. Если это не символическая ссылка или символические ссылки не реализованы в файловой системе, или возникает какая-то системная ошибка, возвращается undef, и необходимо проверить код ошибки, находящийся в переменной $!.

Имейте в виду, что возвращаемая символическая ссылка может быть относительной для указанного вами местоположения. К примеру, можно сказать:

```
$link_contents = readlink("/usr/local/src/express/yourself.h")
```

а readlink может вернуть:

```
../express.1.23/includes/yourself.h
```

что нельзя непосредственно использовать как имя файла, если только текущим каталогом не является */usr/local/src/express*.

readpipe

```
readpipe scalar EXPR
readpipe LIST # (предложение)
```

Внутренняя функция, реализующая конструкцию кавычек qx// (также известную как оператор обратных апострофов). Может пригодиться, когда потребуется задать выражение *EXPR*, которое сложно будет записать в виде строки в кавычках. Учтите, что в будущем мы можем поменять этот интерфейс с целью поддержки аргумента в виде списка, чтобы сделать функцию более похожей на exec, поэтому не рассчитывайте, что она будет по-прежнему предоставлять скалярный контекст для *EXPR*. Укажите scalar сами или попробуйте версию *LIST*. Кто знает, может быть, когда вы это будете читать, она уже заработает.

recv




```
recv SOCKET, SCALAR, LEN, FLAGS
```

Извлекает сообщение из сокета. Пытается скопировать *LENGTH* символов (кодов) в переменную *SCALAR* из дескриптора файла *SOCKET*. Возвращает адрес отправителя или undef, если возникла ошибка. *SCALAR* растет или сокращается в зависимости от количества фактически прочитанных символов. Функция принимает такие же флаги, как *recv*(2) и в действительности реализована с использованием *recvfrom*(2). См. раздел «Сокеты» главы 15.

Обратите внимание на слово *«символов»*: в зависимости от параметров настройки сокета, читаться будут либо байты (восьмибитные значения), либо полностью декодированные символы. По умолчанию все сокеты оперируют байтами. Но, например, если сокет был настроен с помощью binmode на использование фильтра ввода/вывода :encoding(utf8), операции будут выполняться над символами Юникода в кодировке UTF-8, а не над байтами.

redo




```
redo LABEL
redo
```

Оператор redo выполняет переход в начало блока цикла без повторного вычисления условного выражения. Блок continue, если имеется, не выполняется. Если метка *LABEL* опущена, оператор относится к самому внутреннему охватывающему циклу. Этот оператор обычно применяется в программах, которым надо «обмануть» себя в отношении только что введенных данных:

```
# Цикл, объединяющий строчки с обратной косой чертой в конце
while (<STDIN>) {
    if (s/\\\n$// && defined($nextline = <STDIN>)) {
        $_ .= $nextline;
        redo;
    }
    print; # или что-нибудь еще...
}
```

Посредством redo нельзя организовывать выход из блока, возвращающего значение, например eval {}, sub {} или do {}, и его не следует применять для выхода из операции grep или map. Если включен вывод предупреждений, Perl предупредит о применении redo к циклу, находящемуся вне текущей лексической области видимости.

Собственно блок семантически идентичен циклу, выполняемому один раз. Поэтому redo внутри такого блока превращает его в циклическую конструкцию. См. раздел «Управление циклом» главы 4.

ref




```
ref EXPR
ref
```

Оператор ref возвращает истинное значение, если *EXPR* является ссылкой, и ложное в противном случае. Возвращаемое значение зависит от типа того, на что указывает ссылка. Под встроенными типами понимаются следующие:

```
SCALAR
ARRAY
HASH
CODE
REF
GLOB
LVALUE
FORMAT
IO
VSTRING
Regexp
```

Возвращаемое значение LVALUE свидетельствует о том, что ссылка указывает на левостороннее выражение, не являющееся переменной. Подобные ссылки возвращают вызовы таких функций, как pos или substr. Значение VSTRING возвращается, если ссылка указывает на строку с номером версии.

Значение Regexp указывает, что аргумент является регулярным выражением, полученным с помощью qr//.

Если предмет ссылки «освящен» в пакет, возвращается имя этого пакета. Оператор ref можно интерпретировать как оператор «typeof».

```
if (ref($r) eq "HASH") {
    say "r является ссылкой на хеш.";
}
elsif (ref($r) eq "Hump") { # Нехорошо - см. ниже
    say "r является ссылкой на объект Hump.";
}
elsif (not ref $r) {
    say "r вообще не является ссылкой.";
}
```

В ООП считается плохим стилем проверять класс объекта на равенство какому-либо конкретному имени, поскольку производный класс имеет другое имя, но ему разрешен доступ к методам базового класса, согласно принципу подстановки Барбары Лисков (Liskov Substitution Principle, LSP). Лучше использовать метод isa класса UNIVERSAL, как показано ниже:

```
if ($r->isa("Hump") }
    say "r является ссылкой на объект Hump или подкласс.";
}
```

Лучше всего совсем не выполнять проверку, поскольку механизм объектов вообще не отдаст вашему методу объект, не убедившись, что это допустимо. Дополнительные подробности читайте в главах 8 и 12. См. также функцию reftype в разделе с описанием attributes в главе 29.

rename

$! X T

```
rename OLDNAME, NEWNAME
```

Изменяет имя файла. Возвращает истинное значение в случае успеха и ложное в противном случае. Данная функция обычно не переходит границы файловых систем, хотя в системе UNIX это иногда можно компенсировать командой *mv*. Если

файл *NEWNAME* уже существует, он уничтожается. В системах, отличных от UNIX, могут существовать дополнительные ограничения.

Реализовать переименование файлов с перемещением между различными файловыми системами можно с помощью платформо-независимой функции `move` из стандартного модуля `File::Copy`.

require

$_ $! $@ XT

```
require VERSION
require EXPR
require
```

Устанавливает зависимость какого-либо рода от своего аргумента.

Если аргумент является строкой, `require` загружает и выполняет код Perl, находящийся в отдельном файле, имя которого задано строкой. Это похоже на выполнение файла посредством `do`, за исключением того, что `require` проверяет, загружен ли уже библиотечный файл, и возбуждает исключение при возникновении трудностей. (Поэтому ее можно использовать для выражения зависимостей между файлами, не беспокоясь о возможности повторной компиляции.) Как и родственные `do` и `use`, `require` умеет выполнять поиск подключаемых файлов в маршрутах, хранимых в массиве `@INC`, и обновлять `%INC` в случае успеха. См. главу 25.

Файл должен возвращать истинное значение в качестве последнего значения, чтобы сообщить об успешном выполнении кода инициализации, поэтому такие файлы принято заканчивать символами `1;`, если только вы не уверены, что он в любом случае возвращает истинное значение. (Это требование может быть ослаблено в будущем.)

Если аргументом `require` является номер версии вида v5.6.2, `require` требует, чтобы выполняющаяся в данный момент версия Perl была не ниже указанной. (Perl также принимает числа с плавающей запятой, такие как 5.005_03, для совместимости с более старыми версиями Perl, но использование такого формата не поощряется в данное время, поскольку он не понятен людям из других культур.) Поэтому сценарий, которому требуется Perl версии 5.14, может иметь первой строкой:

```
require 5.014_001, # предпочтительнее для обратной совместимости
require 5.14.1;    # то же самое
require v5.14.1;   # проверка версии во время выполнения
```

и более ранние версии Perl прервут свою работу. Однако, как и все другие зависимости, эта проверяется на этапе выполнения. Для проверки во время компиляции можно сказать `use 5.14.0`. См. также описание `$PERL_VERSION` в главе 25.

Если аргументом `require` является голое имя пакета (см. `package`), функция автоматически предполагает наличие суффикса *.pm*, что облегчает загрузку стандартных модулей. Такое поведение похоже на `use`, за исключением того, что происходит это на этапе выполнения, а не на этапе компиляции, и не вызывается метод `import`. Например, чтобы загрузить *Socket.pm*, не вводя символы в текущий пакет, скажите так:

```
require Socket;        # вместо "use Socket;"
```

Однако такого же результата можно добиться следующим кодом, который дополнительно выводит предупреждение на этапе компиляции, если *Socket.pm* не удается найти:

```
use Socket (),
```

Применение require с голым именем также заменяет все :: в имени пакета на системный разделитель каталогов, обычно /. Иными словами, если сказать:

```
require Foo::Bar:       # прекрасное голое имя
```

функция require попытается найти файл *Foo/Bar.pm* в каталогах, перечисленных в массиве @INC. Но если попробовать так:

```
$class = "Foo::Bar"
require $class;         # $class не голое имя
```

или так:

```
require "Foo::Bar";     # литерал в кавычках - не голое имя
```

require будет искать файл *Foo::Bar* в каталогах, перечисленных в массиве @INC, и будет жаловаться, если не сможет его там найти. В этом случае можно сделать так:

```
eval "require $class";
```

Теперь, когда понятно, как require ищет файлы, получив голое слово в виде аргумента, рассмотрим некоторые дополнительные особенности, скрытые от постороннего взгляда. Перед тем, как приступить к поиску файла с расширением *.pm*, функция require сначала попытается найти файл с тем же именем, но с расширением *.pmc*. Если такой файл обнаружится, он загружается вместо файла с расширением *.pm*.

Массив @INC содержит список скаляров, определяющих порядок загрузки модуля. Функция require перебирает элементы этого списка, пока не найдет скалярный элемент, ведущий к загружаемому исходному коду, и затем загружает этот код.

Каждый элемент в @INC должен быть либо строкой (которая интерпретируется как путь к каталогу, где может находиться искомый файл), либо своеобразной «программной сущностью» (используемой для генерации содержимого требуемого файла).

Такой «программной сущностью» может быть ссылка на подпрограмму, массив со ссылкой на подпрограмму (плюс дополнительные аргументы для подпрограммы) или объект с методом INC. Встретив такую «программную сущность», require вызывает ее и передает два аргумента: саму сущность и файл, который требуется отыскать. То есть:

```
Ссылка на подпрограмму:            $sub_ref->($sub_ref, $required_file)
Массив со ссылкой на подпрограмму: $arr_ref->[0]->($arr_ref, $required_file)
Объект:                            $object->INC($required_file)
```

Какая бы форма ни была вызвана, подпрограмма или метод, ожидается, что она вернет список, содержащий до трех значений, которые интерпретируются, как описано в табл. 27.4.

Таблица 27.4. Возвращаемые значения, ожидаемые от ссылок на программный код в @INC

Аргументы	Действие
(*HANDLE*)	Читает исходный код из дескриптора файла *HANDLE*
(*HANDLE*, *CODEREF*)	Читает исходный код из дескриптора файла *HANDLE* и фильтрует через подпрограмму
(*HANDLE*, *CODEREF*, *REF*)	Как и выше, но передает подпрограмме *REF*
(undef, *CODEREF*)	Многократно вызывает подпрограмму, возвращающую строки с исходным кодом
(undef, *CODEREF*, *REF*)	Как и выше, но передает подпрограмме *REF*
Любая другая комбинация	Вызывает ошибку и переход к другому элементу в @INC

Аналогичные обработчики можно также устанавливать в элементы %INC, соответствующие загруженным файлам. См. описание переменной %INC в главе 25.

См. также do *FILE*, команду use, прагму lib и стандартный модуль FindBin.

reset

```
reset EXPR
reset
```

Эту функцию обычно употребляют (или злоупотребляют ею) в начале цикла или в блоке continue в конце цикла с целью очистить глобальные переменные или сбросить скомпилированные операторы поиска m?? в исходное состояние, чтобы их снова можно было использовать. Выражение *EXPR* интерпретируется как список отдельных символов (дефисы могут использоваться для определения диапазонов). Все скалярные переменные, массивы и хеши, начинающиеся с одной из указанных букв, устанавливаются в свое первоначальное состояние. Если выражение опущено, производится сброс операторов однократного поиска соответствия (m?*PATTERN*?), после чего сопоставление может выполняться снова. Сбрасываются только перемененные и операторы поиска в текущем пакете. Всегда возвращает истинное значение.

Чтобы сбросить все переменные с именами, начинающимися символом "X", можно сказать:

```
reset "X";
```

А чтобы сбросить все переменные с именами в нижнем регистре, можно сказать:

```
reset "a-z";
```

Наконец, чтобы сбросить все скомпилированные операторы поиска ??, можно сказать:

```
reset;
```

Сброс "A-Z" в пакете main делать не рекомендуется, поскольку при этом будут очищены глобальные массивы и хеши ARGV, INC, ENV и SIG.

Лексические переменные, создаваемые при помощи my, этой функцией не затрагиваются. Применять reset обычно не рекомендуется, поскольку легко можно

очистить целое пространство имен, и потому что использовать оператор ?? тоже не рекомендуется.

См. также функцию delete_package из стандартного модуля Symbol и описание проблемы в целом в разделе «Защищенные разделы» главы 20.

return



```
return EXPR
return
```

Этот оператор заставляет текущую подпрограмму, eval или do *FILE* немедленно вернуть заданное значение. Попытка использовать return вне трех указанных мест влечет за собой исключение. Обратите также внимание, что eval не может выполнять return от имени подпрограммы, которая вызвала eval.

Выражение *EXPR* может вычисляться в списочном, скалярном или пустом контексте – в зависимости от того, как будет использоваться возвращаемое значение, причем контекст может изменяться от вызова к вызову. Это значит, что передаваемое выражение будет вычисляться в контексте вызова подпрограммы. Если подпрограмма вызывается в скалярном контексте, *EXPR* вычисляется в скалярном контексте. Если подпрограмма вызывается в списочном контексте, *EXPR* вычисляется в списочном контексте и может возвращать список. return без аргументов возвращает скалярное значение undef в скалярном контексте, пустой список () в списочном контексте и (естественно) вообще ничего в пустом контексте. Контекст вызова подпрограммы можно определить, находясь внутри подпрограммы, с помощью (неудачно названной) функции wantarray.

reverse

```
reverse LIST
```

В списочном контексте возвращает список, состоящий из элементов *LIST*, расположенных в обратном порядке. Эту функцию можно использовать для создания убывающих последовательностей:

```
for (reverse 1    10) {     }
```

Поскольку при передаче в списочном контексте хеши превращаются в плоские списки, reverse можно также применять для инвертирования хеша, если допустить, что все значения в нем уникальны:

```
%barfoo = reverse %foobar;
```

В скалярном контексте reverse объединит все элементы *LIST* в одну строку и вернет ее, расположив символы в обратном порядке. Под символами здесь понимаются коды, а не графемы. Это означает, что при неумелом обращении reverse способна превратить все последовательности "\r\n" в "\n\r", а также вызвать перестановку комбинационных символов, отнеся их не к тому базовому символу. Для перестановки графем вместо кодов символов выполните следующее:

```
$codeuni = join "" => reverse $unicode =~ /\X/g;
```

Небольшой совет: обращение списка, отсортированного пользовательской функцией, часто можно упростить, если сразу отсортировать этот список в обратном направлении.

rewinddir



```
rewinddir DIRHANDLE
```

Устанавливает текущую позицию для подпрограммы readdir в *DIRHANDLE* на начало каталога. Эта функция может иметься не во всех системах, поддерживающих readdir, и вызов rewinddir завершается ошибкой, если в системе она не реализована. Возвращает истинное значение в случае успеха и ложное в противном случае.

rindex

```
rindex STR, SUBSTR, POSITION
rindex STR, SUBSTR
```

Работает так же, как index, но возвращает позицию последнего вхождения *SUBSTR* в *STR* (в противоположность index). Возвращает -1, если *SUBSTR* не найдена. Аргумент *POSITION*, если задан, представляет собой самую правую позицию, которую можно вернуть. Выполнить обход строки в обратном направлении можно так:

```
$pos = length $string;
while (($pos = rindex $string, $lookfor, $pos) >= 0) {
    say "Найдена в $pos";
    $pos--;
}
```

Обратите внимание, что, подобно index, функция rindex оперирует позициями символов (кодами), а не позициями графем. Если потребуется интерпретировать строку, как последовательность графем, используйте методы index, rindex и pos из модуля Unicode::GCString, доступного в CPAN.

rmdir



```
rmdir FILENAME
rmdir
```

Удаляет каталог, указанный в *FILENAME*, если он пустой. В случае успеха возвращает истинное значение, в противном случае ложное. См. также модуль File::Path, если вам нужно сначала удалить содержимое каталога и по какой-то причине нежелательно выходить в интерпретатор команд и вызвать *rm -r*. (Например, у вас может не быть интерпретатора или команды *rm*.)

s///



```
s///
```

Оператор подстановки. См. раздел «Операторы поиска по шаблону» главы 5.

say



```
say FILEHANDLE LIST
say FILEHANDLE
say LIST
say
```

Действует так же, как функция print, но неявно добавляет перевод строки в конце. say *LIST* – это более краткая форма записи { local $\ = "\n"; print *LIST* }. При вызове функции с *FILEHANDLE* без списка *LIST* выведет содержимое $_ в заданный дескриптор файла, который должен быть действительным дескриптором файла, таким как FH, а не косвенным, как $fh.

Это ключевое слово доступно, только когда включена особенность "say"; см. раздел «Термы и списочные операторы (влево)» в главе 3.

scalar

```
scalar EXPR
```

Эту псевдофункцию можно использовать внутри *LIST*, чтобы обеспечить принудительное вычисление *EXPR* в скалярном контексте, если вычисление в списочном контексте приводит к другому результату. Например:

```
my ($nextvar) = scalar <STDIN>;
```

не позволяет оператору <STDIN> прочесть все строки из стандартного ввода перед присваиванием, поскольку присваивание списку (даже объявленному с помощью my) создает списочный контекст. (В данном примере без scalar первая строчка из <STDIN> все же была бы присвоена переменной $nextvar, но последующие строчки были бы прочтены и отброшены, поскольку список, которому осуществляется присваивание, может получить только одно скалярное значение.)

Конечно, проще и короче было бы просто отбросить круглые скобки, изменив тем самым списочный контекст на скалярный:

```
my $nextvar = <STDIN>;
```

Поскольку функция say является списочным оператором, необходимо сказать:

```
say "Length is ", scalar(@ARRAY);
```

чтобы вывести длину массива @ARRAY.

Не существует функции list, соответствующей scalar, поскольку на практике никогда не требуется принудительно осуществлять вычисление в списочном контексте. Это связано с тем, что любая операция, где нужен *LIST*, уже бесплатно предоставляет списочный контекст для своих списочных аргументов.

Поскольку scalar является унарным оператором, если по ошибке заключить *EXPR* в круглые скобки, он будет вести себя как скалярное выражение запятой, вычисляя все элементы, кроме последнего, в пустом контексте и возвращая последний элемент, вычисленный в скалярном контексте. Такое поведение редко бывает желаемым. Следующая инструкция:

```
print uc(scalar(&foo,$bar)),$baz;
```

(а)морально эквивалентна таким двум:

```
&foo;
print(uc($bar),$baz);
```

Оператор запятой мы описывали в главе 2. Дополнительно об унарных операторах читайте в разделе «Прототипы» главы 7.

seek



seek *FILEHANDLE*, *OFFSET*, *WHENCE*

Устанавливает позицию в файле для *FILEHANDLE*, точно так же, как вызов *fseek*(3) для стандартного ввода/вывода. Первая позиция в файле имеет смещение 0, а не 1. Кроме того, смещения относятся к позициям байтов, а не символов или строк. В целом, поскольку строки имеют различную длину, невозможно обратиться к строке с конкретным номером, не просмотрев весь файл до этой точки, если только не известно, что все строки имеют определенную длину, или не построен индекс, транслирующий номера строк в байтовые смещения. (Те же ограничения касаются позиций символов в файлах с переменной длиной кодировки символов: операционная система не знает, что такое символы, она понимает только байты.)

FILEHANDLE может быть выражением, возвращающим имя фактического дескриптора файла, переменную typeglob или ссылку на нечто, напоминающее объект дескриптора файла. Возвращает истинное значение в случае успеха и ложное в противном случае. Для удобства может рассчитывать смещения от различных позиций в файле. Значение *WHENCE* задает позицию в файле, используемую в качестве отправной точки для откладывания смещения *OFFSET*: 0 – начало файла; 1 – текущая позиция в файле или 2 – конец файла. Значение *OFFSET* может быть отрицательным при *WHENCE*, равном 1 или 2. Если потребуется использовать для *WHENCE* символические значения, можно воспользоваться константами SEEK_SET, SEEK_CUR и SEEK_END из модуля IO::Seekable, POSIX или Fcntl.

Если понадобится установить позицию в файле для sysread или syswrite, не применяйте seek; буферизация стандартного ввода/вывода делает результат ее действия на позицию в файле непредсказуемым и непереносимым. Используйте sysseek.

Из-за правил и строгостей ANSI C в некоторых системах нужно выполнять seek при каждом переключении между чтением и записью. Кроме того, при этом может вызываться функция библиотеки стандартного ввода/вывода *clearerr* (3). Чтобы избежать смещения позиции в файле, можно задать *WHENCE* равным 1 (SEEK_CUR) и OFFSET равным 0:

```
seek(TEST,0,1);
```

Одно интересное применение этой функции дает возможность обрабатывать растущие файлы, например:

```
for (;;) {
    while (<LOG>) {
        grok($_);      # Обработать текущую строку.
    }
    sleep 15;
    seek LOG,0,1;      # Сбросить ошибку конца файла.
}
```

Последний вызов seek сбрасывает ошибку конца файла, не перемещая текущую позицию в файле. В зависимости от того, насколько стандартной окажется реализация ввода/вывода в конкретной библиотеке C, может понадобиться что-то еще, например:

```
for (;;) {
    for ($curpos = tell FILE; <FILE>; $curpos = tell FILE) {
```

```
        grok($_);              # Обработать текущую строку.
    }
    sleep $for_a_while;
    seek FILE, $curpos, 0;     # Сбросить ошибку конца файла.
}
```

Аналогичная стратегия может применяться для запоминания seek-адреса каждой строки в массиве.

Внимание: аргумент *POSITION* определяет значение позиции в байтах, а не в символах, независимо от того, задействован ли в дескрипторе файла какой-либо фильтр ввода/вывода для декодирования. Однако все функции в Perl, выполняющие чтение файлов, *используют* тот или иной фильтр кодирования/декодирования, и потому есть вероятность прочитать часть «символа» и получить недопустимую строку. Старайтесь не смешивать вызовы sysseek и seek с функциями ввода/вывода при работе с дескрипторами файлов, предусматривающими обработку многобайтных кодировок.

seekdir



```
seekdir DIRHANDLE, POS
```

Устанавливает текущую позицию для следующего обращения к readdir в *DIRHANDLE*. В аргументе *POS* должно передаваться значение, возвращаемое telldir. Для этой функции действуют те же предостережения относительно возможного сжатия каталога, как и для соответствующей подпрограммы системной библиотеки. Функция может быть реализована не везде, где есть readdir. Она наверняка не реализована там, где нет readdir.

select (дескриптор выходного файла)



```
select FILEHANDLE
select
```

В силу исторических причин есть два оператора select, совершенно не связанных друг с другом. О втором рассказывается в следующем разделе. Данная версия оператора select возвращает выбранный в настоящее время дескриптор выходного файла и, если задан *FILEHANDLE*, устанавливает текущий дескриптор файла для вывода. Это влечет два результата: во-первых, write или print без дескриптора файла по умолчанию осуществляют вывод в этот *FILEHANDLE*. Во-вторых, специальные переменные, связанные с выводом, будут ссылаться на этот дескриптор выходного файла. Например, если потребуется установить одинаковый формат верхнего колонтитула сразу для нескольких дескрипторов вывода, можно поступить так:

```
select REPORT1;
$^ = "MyTop";
select REPORT2;
$^ = "MyTop";
```

Но обратите внимание, что при этом REPORT2 остается текущим выбранным дескриптором файла. Такое поведение можно рассматривать как антисоциальное, поскольку оно способно повлиять на работу функций print или write в других подпрограммах. Правильно написанные библиотечные подпрограммы оставляют на выходе текущий дескриптор файла таким же, каким он был на входе. Чтобы под-

держивать такой режим, *FILEHANDLE* можно сделать выражением, возвращающим имя фактического дескриптора файла. Сохранять и восстанавливать текущий выбранный дескриптор файла можно таким образом:

```
my $oldfh = select STDERR;
$| = 1;
select $oldfh.
```

или идиоматически, но несколько менее прозрачно:

```
select((select(STDERR), $| = 1)[0])
```

Этот пример создает список из значений, возвращаемых вызовом select(STDERR) (который в качестве побочного эффекта выбирает STDERR), и $| = 1 (что всегда дает 1), но устанавливает автоматическую очистку буфера для назначаемого текущим дескриптором STDERR в качестве побочного эффекта. Первый элемент списка (прежний выбранный дескриптор файла) используется теперь в качестве аргумента для внешнего оператора select. Причудливо, а? Вот что получается, когда твои знания Lisp делают тебя опасным человеком.

Можно также использовать стандартный модуль SelectSaver, чтобы автоматически восстанавливать прежний select после выхода из области видимости.

Однако после всего сказанного следует заметить, что в наше время редко требуется применять такую разновидность select, ведь для большинства специальных переменных, которые может потребоваться установить, существуют объектно-ориентированные методы-обертки, которые сделают это автоматически. Поэтому вместо непосредственного присваивания $| можно сказать:

```
use IO::Handle;            # К сожалению, это *не* маленький модуль.
STDOUT->autoflush(1);
```

А предыдущий пример форматирования можно записать так:

```
use IO::Handle;
REPORT1->format_top_name("MyTop");
REPORT2->format_top_name("MyTop");
```

select (готовые дескрипторы файлов)




```
select RBITS, WBITS, EBITS, TIMEOUT
```

Оператор select с четырьмя аргументами совершенно не связан с описанным выше оператором select. Этот оператор используется, чтобы определить дескрипторы файлов, готовые для ввода или вывода, или чтобы сообщить о наличии исключительной ситуации. (Благодаря этому можно избежать необходимости проводить опрос.) Этот оператор обращается к системному вызову *select*(2) с указанными битовыми масками, которые можно создать с помощью fileno и vec, например:

```
$rin = $win = $ein = "";
vec($rin, fileno(STDIN), 1) = 1;
vec($win, fileno(STDOUT), 1) = 1;
$ein = $rin | $win;
```

Чтобы выполнить select для нескольких дескрипторов файлов, можно написать такую подпрограмму:

```
sub fhbits {
    my @fhlist = @_;
    my $bits;
    for my $fh (@fhlist) {
        vec($bits, fileno($fh), 1) = 1
    }
    return $bits;
}
$rin = fhbits(*STDIN, *TTY, *MYSOCK);
```

Обратите внимание, что здесь дескрипторы файлов передаются функции в виде ссылок на их typeglobs, потому что передача их в виде строк – не самое лучшее решение. Если вы собираетесь использовать «самооживляемые» дескрипторы файлов, не делайте этого.

Для повтоного использования битовых масок (а это более эффективно) обычно используется идиома:

```
($nfound, $timeleft) =
    select($rout=$rin, $wout=$win, $eout=$ein, $timeout);
```

Или осуществляется блокировка, пока не станет доступен какой-нибудь дескриптор файла:

```
$nfound = select($rout=$rin, $wout=$win, $eout=$ein, undef);
```

Как видите, вызов select в скалярном контексте просто возвращает $nfound – число готовых дескрипторов.

Значением присваивания $wout=$win является его левая часть, поэтому сначала $wout получает новое значение, а затем вызывается select, в то время как $win остается неизменной.

В любых аргументах можно передать undef, и тогда они игнорируются. Значение *TIMEOUT*, если оно не равно undef, выражается в секундах и может быть дробным. (Равный 0 интервал ожидания *TIMEOUT* действует как опрос.) Далеко не все реализации способны возвращать $timeleft. Неспособные на это возвращают значение $timeleft, равное переданному $timeout.

Стандартный модуль IO::Select предоставляет более дружественный интерфейс к select, в основном благодаря тому, что берет на себя всю работу по созданию битовой маски.

Одно из применений select состоит в создании пауз большего разрешения, чем позволяет sleep. Следует указать undef для всех битовых масок. Чтобы приостановить выполнение на (по крайней мере) 4,75 секунды используйте:

```
select undef, undef, undef, 4.75;
```

(В некоторых системах, отличных от UNIX, тройной undef может не работать, и тогда потребуется подделать по крайней мере одну битовую маску действующего дескриптора, который никогда не будет готов.)

В настоящее время предпочтительнее импортировать специальную версию sleep из стандартного модуля Time::HiRes, обеспечивающего более высокую переносимость:

```
use Time::HiRes qw(sleep);
sleep 4.75;         # не обычный sleep
```

Вряд ли следует смешивать буферизованный ввод/вывод (как в случае read или <HANDLE>) и select, кроме случаев, разрешенных POSIX, и даже в этом случае только на подлинных системах POSIX. Применяйте вместо этого sysread.

semctl



```
semctl ID, SEMNUM, CMD, ARG
```

Вызывает функцию System V IPC *semctl*(2). Возможно, придется сначала сказать use IPC::SysV, чтобы получить определения констант. Если *CMD* имеет значение IPC_STAT или GETALL, аргумент ARG должен быть переменной, содержащей возвращаемую структуру semid_ds или массив значений семафоров. Подобно ioctl и fcntl, возвращает undef в случае ошибки, "0 but true" обозначает ноль и фактическое значение в противном случае.

См. также модуль IPC::Semaphore. Данная функция доступна только в системах, поддерживающих System V IPC.

semget



```
semget KEY, NSEMS, FLAGS
```

Обращается к системному вызову System V IPC *semget*(2). Перед вызовом следует выполнить use IPC::SysV, чтобы получить определения констант. Возвращает идентификатор семафора или undef в случае ошибки.

См. также модуль IPC::Semaphore. Данная функция доступна только в системах, поддерживающих System V IPC.

semop

```
semop KEY, OPSTRING
```

Обращается к системному вызову System V IPC *semop*(2), чтобы выполнить такие операции с семафорами, как передача сигналов и ожидание. Перед вызовом следует выполнить use IPC::SysV, чтобы получить определения констант.

Аргумент *OPSTRING* должен быть упакованным массивом структур semop. Структуру semop можно создать вызовом pack("s*", $semnum, $semop, $semflag). Число операций с семафорами определяется длиной *OPSTRING*. Функция возвращает истинное значение в случае успеха и ложное, если возникла ошибка.

Следующий код ожидает на семафоре $semnum с идентификатором в $semid:

```
$semop = pack "s*", $semnum, -1, 0;
semop($semid, $semop)    || die "Semaphore trouble: $!"
```

Для передачи сигнала семафору просто замените -1 на 1.

См. раздел «System V IPC» главы 15. См. также модуль IPC::Semaphore. Данная функция доступна только в системах, поддерживающих System V IPC.

send



```
send SOCKET, MSG, FLAGS, TO
send SOCKET, MSG, FLAGS
```

Посылает сообщение в сокет. Принимает те же флаги, что и одноименный системный вызов, – см. *send*(2). Для несоединенных сокетов нужно указать адрес отправки *TO*, что заставляет функцию Perl send работать как *sendto*(2). Системный вызов *sendmsg*(2) в настоящее время не реализован в стандартном Perl. Функция send возвращает число отправленных символов или undef, если возникла ошибка.

Обратите внимание на слово «*символов*»: в зависимости от параметров настройки сокета, отправляться будут либо байты (восьмибитные значения), либо символы. По умолчанию все сокеты оперируют байтами. Но, например, если сокет был настроен с помощью binmode на использование фильтра ввода/вывода :encoding(utf8), операции будут выполняться над символами Юникода в кодировке UTF-8, а не над байтами.

(Некоторые системы, отличные от UNIX, неверно рассматривают сокеты как отличные от обычных дескрипторов файлов, в результате чего всегда приходится использовать для сокетов send и recv вместо более удобных операторов стандартного ввода/вывода.)

Ошибка, которую часто совершает как минимум один из нас, состоит в том, что путаются send Perl и send C в записи:

```
send SOCK, $buffer, length $buffer     # НЕВЕРНО
```

Этот код непонятным образом приводит к ошибке в зависимости от связи между длиной строки и битами FLAGS, которые ожидает система. Примеры см. в разделе «Передача сообщений» главы 15.

setpgrp

$! | X T | X U

setpgrp *PID*, *PGRP*

Устанавливает текущую группу процессов (PGRP) для процесса с указанным *PID* (для текущего процесса используйте *PID*, равный 0). Вызов setpgrp возбудит исключение при вызове в системе, не поддерживающей *setpgrp*(2). Будьте осторожны: некоторые системы игнорируют передаваемые аргументы и всегда выполняют setpgrp(0, $$). К счастью, именно эти аргументы обычно и требуется передать. Если аргументы опущены, по умолчанию они равны 0,0. В операционной системе BSD версии 4.2 функция setpgrp не принимала никаких аргументов, но в BSD 4.4 это синоним для функции setpgid. Для лучшей переносимости (согласно некоторому ее определению) используйте функцию setpgid из модуля POSIX непосредственно. Если в действительности вы пытаетесь превратить свой сценарий в программу-демон, стоит подумать также о применении функции POSIX::setsid. Обратите внимание, что POSIX-версия setpgrp не принимает аргументы, поэтому только setpgrp(0,0) является действительно переносимой.

setpriority

$! | X T | X U

setpriority *WHICH*, *WHO*, *PRIORITY*

Устанавливает текущий приоритет *PRIORITY* для процесса, группы процессов или пользователя в соответствии с *WHICH* и *WHO*. См. *setpriority*(2). Возбуждает исключение при вызове в системе, не поддерживающей *setpriority*(2). Чтобы понизить приоритет процесса на четыре единицы (как при запуске программы с *nice*(1)), сделайте так:

```
setpriority 0, 0, getpriority(0, 0) + 4;
```

В разных системах значение приоритета может интерпретироваться по-разному. Некоторые приоритеты могут быть недоступны для непривилегированных пользователей.

См. также модуль BSD::Resource в CPAN.

setsockopt



```
setsockopt SOCKET, LEVEL, OPTNAME, OPTVAL
```

Устанавливает параметры сокета. Возвращает undef при ошибке. Все константы, необходимые для передачи в аргументах *LEVEL* и *OPTNAME*, определены в модуле Socket. Значения для аргумента *LEVEL* можно также получить с помощью функции getprotobyname. *LEVEL* указывает, на какой уровень протокола нацелен вызов, или SOL_SOCKET – для самого сокета поверх всех уровней. Значением аргумента *OPTVAL* может быть либо упакованная строка, либо целое число. Целочисленное значение *OPTVAL* является более короткой формой записи pack("i", *OPTVAL*). *OPTVAL* можно задать как undef, если вы не хотите передавать аргумент.

Часто для сокета устанавливается параметр SO_REUSEADDR, чтобы обойти проблему, возникающую при невозможности привязаться к конкретному адресу, когда предыдущее соединение TCP на этом порту все еще принимает решение о закрытии. Это выглядит так:

```
use Socket;
socket(SOCK, ...) || die "Невозможно создать сокет: $!\n"
setsockopt(SOCK, SOL_SOCKET, SO_REUSEADDR, 1)
        || warn "Невозможно выполнить setsockopt: $!\n"
```

Другим типичным параметром является запрет использования алгоритма Нейгла (Nagle):

```
use Socket qw(IPPROTO_TCP TCP_NODELAY),
setsockopt($socket, IPPROTO_TCP, TCP_NODELAY, 1);
```

См. в *setsockopt*(2) другие возможные значения.

shift



```
shift ARRAY
shift
```

Сдвигает первое значение массива и возвращает его, укорачивая массив на единицу и смещая все вниз. (Или вверх, в зависимости от того, как вы зрительно представляете себе массив. Мы предпочитаем влево.) Если в массиве нет элементов, функция возвращает undef.

Если аргумент *ARRAY* опущен, функция сдвигает @_ в лексической области видимости подпрограмм и форматов; она сдвигает @ARGV в областях видимости файлов (обычно основной программы) или в лексических областях видимости, установленных конструкциями eval *STRING*, BEGIN {}, CHECK {}, UNITCHECK {}, INIT {} и END {}.

Подпрограммы часто начинают работу с того, что копируют свои аргументы в лексические переменные, используя `shift`:

```
sub marine {
    my $fathoms = shift; # глубина
    my $fishies = shift; # число рыб
    my $o2      = shift; # содержание кислорода
    # ...
}
```

Также `shift` применяется для обработки аргументов в начале программы:

```
while (defined($_ = shift)) {
    /^[^-]/     && do { unshift @ARGV, $_; last };
    /^-w/       && do { $WARN = 1;         next };
    /^-r/       && do { $RECURSE = 1       next };
    die "Неизвестный аргумент $_";
}
```

Для обработки аргументов программ предпочтительнее использовать модули `Getopt::Std` и `Getopt::Long`.

Начиная с версии v5.14, функция `shift` может принимать ссылку на «неосвященный» массив, которая будет разыменована автоматически. Эта особенность `shift` считается экспериментальной. Ее поведение может измениться в будущих версиях Perl.

См. также функции `unshift`, `push`, `pop` и `splice`. Функции `shift` и `unshift` делают с левым концом массива то же самое, что `pop` и `push` с правым.

shmctl

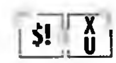
`shmctl` *`ID, CMD, ARG`*

Вызывает системную функцию System V IPC *shmctl*(2). Перед вызовом следует выполнить `use IPC::SysV`, чтобы загрузить определения констант.

Если *`CMD`* имеет значение `IPC_STAT`, *`ARG`* должен быть переменной, в которую будет помещена возвращаемая структура `shmid_ds`. Подобно `ioctl` и `fcntl`, возвращает `undef` в случае ошибки, `"0 but true"` как обозначение нуля, и фактическое значение в остальных случаях.

Доступна только в системах, поддерживающих System V IPC.

shmget



`shmget` *`KEY, SIZE, FLAGS`*

Вызывает системную функцию System V IPC *shmget*(2). Возвращает идентификатор сегмента совместно используемой памяти или `undef` в случае ошибки. Перед вызовом следует выполнить `use SysV::IPC`.

Доступна только в системах, поддерживающих System V IPC.

shmread



`shmread ID, VAR, POS, SIZE`

Осуществляет чтение из сегмента совместно используемой памяти ID, начиная с позиции *POS* размером *SIZE* (подключаясь, копируя и отключаясь от него). Аргумент *VAR* должен быть переменной, куда функция поместит прочитанные данные. Возвращает истинное значение в случае успеха и ложное в случае ошибки.

Доступна только в системах, поддерживающих System V IPC.

shmwrite



`shmwrite ID, STRING, POS, SIZE`

Осуществляет запись в сегмент совместно используемой памяти ID, начиная с позиции *POS* размером *SIZE* (подключаясь, копируя и отключаясь путем прикрепления к нему, копирования в него данных и открепления от него). Если *STRING* длиннее *SIZE*, скопировано будет только *SIZE* байтов; если *STRING* короче *SIZE*, недостающие байты будут заполнены нулями. Возвращает истинное значение в случае успеха и ложное в случае ошибки.

Доступна только в системах, поддерживающих System V IPC. (Вы, наверное, уже устали это читать, а мы устали это повторять.)

shutdown



`shutdown SOCKET, HOW`

Закрывает соединение с сокетом способом, указанным в аргументе *HOW*. Если *HOW* имеет значение 0, дальнейший прием данных запрещен. Если *HOW* имеет значение 1, дальнейшая отправка данных запрещена. Если *HOW* имеет значение 2, запрещено все.

```
shutdown(SOCK, 0); # больше не читать
shutdown(SOCK, 1); # больше не писать
shutdown(SOCK, 2); # больше никакого ввода/вывода
```

Это удобно при работе с сокетами, когда необходимо сообщить другой стороне, что закончена запись, но не закончено чтение, или наоборот. Это также более настойчивая форма закрытия, поскольку отключает также все копии дескрипторов файлов, имеющиеся в ответвившихся процессах.

Представьте сервер, которому нужно читать запрос клиента до тех пор, пока не будет получен конец файла, а затем отправить ответ. Если клиент вызовет `close`, сокет окажется недоступным для ввода/вывода, поэтому получить ответ будет невозможно. Вместо этого клиент должен использовать `shutdown` и «полузакрыть» соединение:

```
say SERVER "my request";    # отправить какие-то данные
shutdown(SERVER, 1);        # послать eof; больше не записывать
$answer = <SERVER>;         # но можно продолжать чтение
```

(Если вы пришли сюда, чтобы узнать, как остановить систему, вам придется для этого выполнить внешнюю программу. См. `system`.)

sin

$_

```
sin EXPR
sin
```

Извините, ничего греховного[1] в этом операторе нет. Он просто возвращает синус выражения *EXPR* (выраженного в радианах).

Для выполнения операции, обратной синусу, можно вызвать функцию asin из модуля Math::Trig или POSIX, либо использовать следующее соотношение:

```
sub asin { atan2($_[0], sqrt(1 - $_[0] * $_[0])) }
```

sleep

```
sleep EXPR
sleep
```

Заставляет сценарий уснуть (сделать паузу) на *EXPR* секунд либо навсегда, если аргумент *EXPR* опущен, и возвращает число истекших секунд паузы. Паузу можно прервать, послав процессу SIGALRM. В некоторых старых системах пауза может продолжаться на одну секунду меньше, чем запрошено, в зависимости от способа подсчета секунд. Большинство современных систем точно выдерживает паузу. Однако может оказаться, что пауза продолжится дольше, чем нужно, поскольку в нагруженной многозадачной системе планирование данного процесса может задержаться. Для определения продолжительности паузы с более высокой точностью, чем одна секунда, можно воспользоваться функцией usleep из стандартного модуля Time::HiRes. Если в системе имеется вызов select (готовых дескрипторов файлов), с его помощью можно получить более высокое разрешение времени. В некоторых системах UNIX можно также использовать syscall для вызова *getitimer*(2) и *setitimer*(2). Чередование вызовов alarm и sleep может оказаться недопустимым, поскольку sleep часто реализуется с помощью alarm.

См. также функцию pause в модуле POSIX.

socket




```
socket SOCKET, DOMAIN, TYPE, PROTOCOL
```

Открывает сокет заданного вида и связывает его с дескриптором файла *SOCKET*. *DOMAIN*, *TYPE* и *PROTOCOL* задаются так же, как в *socket*(2). Если *SOCKET* не определен, он «самооживляется». Вызову этой функции в программе должна предшествовать строка:

```
use Socket;
```

Она поставляет нужные константы. Функция возвращает истинное значение в случае успешного выполнения. Примеры см. в разделе «Сокеты» главы 15.

В системах, которые поддерживают для файлов флаг закрытия при выполнении, этот флаг устанавливается для вновь открытого дескриптора файла в соответствии со значением $^F. См. переменную $^F ($SYSTEM_FD_MAX) в главе 25.

---

[1] Слово sin, являясь сокращением математического термина «синус», также переводится с английского языка, как «грех». – *Прим. перев.*

socketpair



```
socketpair SOCKET1, SOCKET2, DOMAIN, TYPE, PROTOCOL
```

Создает неименованную пару сокетов указанного типа в указанном домене. *DOMAIN*, *TYPE* и *PROTOCOL* задаются так же, как в *socketpair*(2). Для получения необходимых констант используйте `use Socket`. **Если какой-либо из аргументов *SOCKET1* и *SOCKET2* имеет неопределенное значение, он будет «самооживлен». Функция возвращает истинное значение в случае успеха и ложное в противном случае. В системах, где вызов *socketpair*(2) не реализован, эта функция возбуждает исключение.**

Данная функция обычно вызывается непосредственно перед `fork`. **Один из порожденных процессов должен закрыть *SOCKET1*, а второй – *SOCKET2*. Эти сокеты можно использовать двунаправленным образом, в отличие от дескрипторов файлов, создаваемых функцией** `pipe`. **В некоторых системах** `pipe` **определяется через** `socketpair`, **и тогда вызов** `pipe(Rdr, Wtr)` **является, в сущности, такой последовательностью инструкций:**

```
use Socket;
socketpair(Rdr, Wtr, AF_UNIX, SOCK_STREAM, PF_UNSPEC);
shutdown(Rdr, 1);        # запись читающему процессу запрещена
shutdown(Wtr, 0);        # чтение пишущему процессу запрещено
```

В Perl 5.8 и выше действие функции имитируется с применением IP-сокетов для хоста localhost, если система реализует сокеты, но не через `socketpair`. **В системах, поддерживающих для файлов флаг закрытия при выполнении, этот флаг устанавливается для вновь открытого дескриптора файла в соответствии со значением** `$^F`. **См. переменную** `$^F` (`$SYSTEM_FD_MAX`) **в главе 25. См. также пример в конце раздела «Двунаправленная связь» главы 15.**

sort



```
sort USERSUB LIST
sort BLOCK LIST
sort LIST
```

Сортирует список *LIST* и возвращает отсортированное списочное значение. По умолчанию сортирует в стандартном порядке сравнения строк (с использованием, возможно перегруженного, оператора `cmp`). **Для сортировки в лексикографическом порядке следует использовать модуль** `Unicode::Collate`; **см. раздел «Сравнение и сортировка строк Юникода» в главе 6. Проще говоря, самый легкий способ реализовать сортировку в алфавитном порядке выглядит так:**

```
use Unicode::Collate;
@alphabetized_list = Unicode::Collate->new->sort(@list);
```

Если действует прагма `locale`, **вызов** `sort` *LIST* **отсортирует *LIST* в соответствии с текущими национальными настройками. Даже если такие национальные настройки существуют, Perl не поддерживает настройки с многобайтными символами, поэтому подобный подход едва ли позволит получить желаемый результат. Если требуется обеспечить надежную сортировку в национальных алфавитах, используйте модуль** `Unicode::Collate::Locale`.

***USERSUB*, если присутствует, представляет имя подпрограммы, возвращающей целое число, меньшее, равное или большее 0, в зависимости от того, как должны**

быть упорядочены элементы списка. (Для трехстороннего сравнения чисел и строк можно использовать удобные операторы <=> и cmp.) Если аргумент *USERSUB* задан, но функция не определена, sort возбудит исключение.

С целью повышения производительности вызов подпрограммы выполняется в обход обычной процедуры вызова, а потому подпрограмма не должна быть рекурсивной (нельзя также выходить из блока или подпрограммы с помощью операторов управления циклом), сравниваемые элементы передаются в подпрограмму не через @_, а временно присваиваются глобальным переменным $a и $b пакета, в котором была скомпилирована sort (см. примеры ниже). Переменные $a и $b являются псевдонимами фактических значений, поэтому не изменяйте их в подпрограмме.

Подпрограмма сравнения должна корректно вести себя. Если она действует непоследовательно (например, иногда сообщая, что $x[1] меньше $x[2], а иногда утверждая обратное), результат сортировки будет сложно предсказать. (Это еще одна причина, почему не следует модифицировать $a и $b.)

Аргумент *USERSUB* может быть именем скалярной переменной (без индексов), и тогда он задает символическую или жесткую ссылку на подпрограмму, которая должна использоваться. (Символическое имя, но не жесткая ссылка, допускается, даже когда действует директива use strict 'refs'.) Вместо *USERSUB* можно задать блок в качестве анонимной внедряемой (inline) подпрограммы сортировки.

Чтобы выполнить обычную сортировку чисел, скажите:

```
sub numerically { $a <=> $b }
@sortedbynumber = sort numerically 53,29,11,32,7;
```

Чтобы выполнить сортировку в порядке убывания, можно просто применить после сортировки reverse либо поменять местами $a и $b в подпрограмме сортировки:

```
@descending = reverse sort numerically 53,29,11,32,7;

sub reverse_numerically { $b <=> $a }
@descending = sort reverse_numerically 53,29,11,32,7;
```

Чтобы выполнить сортировку ASCII-строк без учета регистра, перед сравнением пропустите $a и $b через lc:

```
@unsorted = qw/sparrow Ostrich LARK catbird blueJAY/;
@sorted = sort { lc($a) cmp lc($b) } @unsorted;
```

В случае работы с Юникодом (в отличие от ASCII) ни lc, ни uc не могут использоваться для канонизации регистра, поскольку отношения между символами в разных регистрах имеют более сложную форму, чем могут выразить эти две функции. В настоящее время различаются три регистра символов, а не два, и между ними нет прямого, однозначного соответствия – например, некоторым символам в верхнем регистре соответствует несколько символов в нижнем регистре, и наоборот. Ожидается, что для решения этой проблемы в Perl когда-нибудь появится поддержка функции fc, названной так, потому что она производит «свертку регистра» (casefold), как это делает модификатор /i шаблонов. Функция fc должна появиться примерно в Perl версии v5.16, возможно, в виде use feature "fc". Если функция fc будет доступна – используйте ее вместо lc в своих алгоритмах сортировки, использующих cmp, за исключением случаев, когда сортируемый текст весьма замысловатый или вы желаете выполнить сортировку по числовым значе-

ниям кодов символов. Если функция fc недоступна или требуется отсортировать текст в алфавитном (словарном) порядке, а не по кодам символов, читайте раздел «Сравнение и сортировка строк Юникода» в главе 6.

Функция sort часто применяется для сортировки хешей по значениям. Например, если хеш %sales_amount содержит записи объемов продаж по подразделениям, выполнение в программе сортировки поиска в хеше позволяет отсортировать ключи по соответствующим им значениям:

```
# сортировать подразделения по объемам продаж в порядке убывания
sub bysales { $sales_amount{$b} <=> $sales_amount{$a} }

for $dept (sort bysales keys %sale_amount) {
    say "$dept => $sales_amount{$dept}";
}
```

Можно создавать дополнительные уровни сортировки, образуя каскады из нескольких сравнений с помощью операторов || или or. Это возможно благодаря тому, что операторы сравнения возвращают 0 для эквивалентных значений, благодаря чему выполнение «проваливается» к следующему сравнению. В следующем примере ключи хеша сортируются сначала по объемам продаж, а затем по самим ключам (в случае, если два или более подразделений имеют одинаковый объем продаж):

```
sub by_sales_then_dept {
    $sales_amount{$b} <=> $sales_amount{$a}
                       ||
                  $a cmp $b
}

for $dept (sort by_sales_then_dept keys %sale_amount) {
    say "$dept => $sales_amount{$dept}";
}
```

Предположим, что @recs представляет собой массив ссылок на хеши, и каждый хеш содержит поля FIRSTNAME, LASTNAME, AGE, HEIGHT и SALARY. Следующий код отсортирует записи о людях, перемещая вперед тех, кто богаче, затем выше, затем моложе, затем в алфавитном порядке имен:

```
sub prospects {
    $b->{SALARY}    <=> $a->{SALARY}
        ||
    $b->{HEIGHT}    <=> $a->{HEIGHT}
        ||
    $a->{AGE}       <=> $b->{AGE}
        ||
    $a->{LASTNAME}  cmp $b->{LASTNAME}
        ||
    $a->{FIRSTNAME} cmp $b->{FIRSTNAME}
}

@sorted = sort prospects @recs;
```

Основой для сравнения в процедуре сортировки может стать любая полезная информация, которую можно извлечь из $a и $b. Например, если строки текста

должны быть отсортированы согласно определенным полям, для извлечения полей при сортировке можно использовать функцию split.

```
@sorted_lines = sort {
    @a_fields = split /:/, $a;     # поля, разделяемые двоеточиями
    @b_fields = split /:/, $b;

    $a_fields[3] <=> $b_fields[3] # числовая сортировка по полю #4, затем
        ||
    $a_fields[0] cmp $b_fields[0] # строковая сортировка по полю #1, затем
        ||
    $b_fields[2] <=> $a_fields[2] # числовая сортировка в порядке убывания
        ||                        # по полю #3
    ...                           # и так далее
} @lines;
```

Однако поскольку sort выполняет подпрограмму сортировки много раз с различными комбинациями значений $a и $b, в приведенном примере каждая строка будет расщепляться многократно.

Чтобы избежать затрат на повторное выполнение таких действий, как расщепление строк для сравнения полей, действие по извлечению данных можно выполнить один раз перед сортировкой и сохранить полученную информацию. Вот как создаются анонимные массивы для хранения каждой строки вместе с результатами ее расщепления:

```
@temp = map { [$_, split /:/] } @lines;
```

После этого сортируются ссылки на массивы:

```
@temp = sort {
    @a_fields = @$a[1..$#$a];
    @b_fields = @$b[1..$#$b];

    $a_fields[3] <=> $b_fields[3] # числовая сортировка по полю #4, затем
        ||
    $a_fields[0] cmp $b_fields[0] # сортировка строк по полю #1, затем
        ||
    $b_fields[2] <=> $a_fields[2] # числовая сортировка в порядке убывания
        ||                        # по полю #3
    ...                           # и так далее
} @temp;
```

После сортировки ссылок на массивы из них можно извлечь исходные строки:

```
@sorted_lines = map { $_->[0] } @temp;
```

Данный прием map-sort-map, который часто называют преобразованием Шварца (Schwartzian Transform)[1], может быть выполнен в одну инструкцию:

```
@sorted_lines = map { $_->[0] }
                sort {
                    $a->[4] <=> $b->[4] # осторожно: индексы действительно
                                        # начинаются с 1
```

[1] В честь ее автора – Рэндала Шварца. Описывается, например, на странице *http://members.home.net/andrew-johnson/perl/archit/msg00041.html*. – *Прим. ред.*

```
                    ||
                $a->[1] cmp $b->[1]
                    ||
                $a->[3] <=> $b->[3]
                    ||
                ...
            }
            map { [$_, split /:/] } @lines;
```

Не объявляйте $a и $b как лексические переменные (с помощью my). Это – глобальные переменные пакета (хотя на них не распространяются обычные ограничения глобальных переменных при использовании use strict). Однако подпрограмма сортировки должна находиться в том же пакете, либо квалифицировать $a и $b именем пакета вызывающей подпрограммы.

Ко всему этому добавим, что *можно* писать подпрограммы сортировки с обычным способом передачи аргументов (и, что не случайно, использовать подпрограммы XS в качестве подпрограмм сортировки) при условии, что подпрограмма сортировки объявляется с прототипом ($$). В этом случае $a и $b можно объявлять как лексические переменные:

```
sub numerically ($$) {
    my ($a, $b) = @_;
    $a <=> $b;
}
```

Когда-нибудь, когда будут реализованы полные прототипы, вы сможете просто сказать:

```
sub numerically ($a, $b) { $a <=> $b }
```

и тогда мы вернемся к тому, с чего начали. Более или менее.

В Perl v5.6 и более ранних версий сортировка реализована с использованием алгоритма *быстрой сортировки* (*quicksort*). Этот алгоритм не устойчив и *может* показывать квадратичную сложность. (Устойчивый алгоритм сортировки должен сохранять исходный порядок следования элементов, если они равны. Хотя в среднем по всем массивам длины n производительность алгоритма быстрой сортировки составляет $O(n \log n)$, иногда его производительность может снижаться до $O(n^2)$.) В экспериментальной версии v5.7 вместо алгоритма быстрой сортировки используется реализация устойчивого алгоритма сортировки слиянием (mergesort), который в худшем случае имеет производительность $O(n \log n)$. Однако тесты показывают, что при некоторых исходных данных на некоторых платформах исходный алгоритм быстрой сортировки работает быстрее. В Perl версии v5.8 появилась прагма sort, обеспечивающая ограниченные возможности управления сортировкой. Вероятно, этот не совсем полноценный контроль над алгоритмом сортировки исчезнет в будущих версиях Perl, но возможность охарактеризовать ввод или вывод, скорее всего, останется. См. раздел «sort» в главе 29.

splice




```
splice ARRAY, OFFSET, LENGTH, LIST
splice ARRAY, OFFSET, LENGTH
splice ARRAY, OFFSET
splice ARRAY
```

Удаляет элементы, задаваемые *OFFSET* и *LENGTH*, из *ARRAY* и заменяет их элементами из *LIST*, если он задан. Если смещение *OFFSET* отрицательно, отсчет начинается с конца массива, но если получившаяся позиция окажется перед началом массива, возникает исключение. В списочном контексте splice возвращает элементы, удаленные из массива. В скалярном контексте возвращает последний удаленный элемент или undef, если такого не было. Если число новых элементов не равно числу прежних элементов, массив при необходимости расширяется или сокращается, а индексы элементов изменяются соответственно. Если длина *LENGTH* опущена, функция удаляет все до конца, начиная с *OFFSET*. Если опущены оба аргумента, *OFFSET* и *LENGTH*, функция удалит все элементы из массива. Если позиция *OFFSET* окажется за концом массива, Perl выведет предупреждение, а список *LIST* будет добавлен в конец массива.

Эквиваленты перечислены в табл. 27.5.

Таблица 27.5. Эквиваленты функции splice при работе с массивами

Прямой метод	Эквивалент splice
push(@a, $x, $y)	splice(@a, @a, 0, $x, $y)
pop(@a)	splice(@a, -1)
shift(@a)	splice(@a, 0, 1)
unshift(@a, $x, $y)	splice(@a, 0, 0, $x, $y)
$a[$x] = $y	splice(@a, $x, 1, $y)
(@a, @a = ())	splice(@a)

Функцию splice удобно также применять для разрезания списка аргументов, переданных подпрограмме. Допустим, например, что перед списками передаются их длины:

```
sub list_eq { # сравнить два списочных значения
    my @a = splice(@_, 0, shift);
    my @b = splice(@_, 0, shift);
    return 0 unless @a == @b;          # одинаковая длина?
    while (@a) {
        return 0 if pop(@a) ne pop(@b);
    }
    return 1;
}
if (list_eq($len, @foo[1..$len], scalar(@bar), @bar)) {     }
```

Однако лучше использовать для этого ссылки на массивы.

Начиная с версии v5.14 splice может принимать ссылку на «неосвященный» хеш или массив, которая будет разыменована автоматически. Эта особенность splice считается экспериментальной. Ее поведение может измениться в будущем.

split




```
split /PATTERN/, EXPR, LIMIT
split /PATTERN/, EXPR
split /PATTERN/
split
```

Просматривает строку *EXPR* в поисках разделителей и расщепляет ее на список подстрок, возвращая в списочном контексте полученный список, а в скалярном контексте число подстрок.[1] Разделители находятся путем многократного поиска по шаблону с использованием регулярного выражения *PATTERN*, поэтому разделители могут иметь любой размер и не обязаны быть одной и той же строкой в каждом поиске. (Обычно разделители не возвращаются; исключения из этого правила излагаются ниже.) Если шаблон не удается отыскать в строке, split возвращает исходную строку в качестве единственной подстроки. При нахождении одного соответствия возвращаются две подстроки и т. д. В шаблоне можно использовать модификаторы регулярных выражений, например */PATTERN/*i, */PATTERN/*x и тому подобные. В случае расщепления по шаблону /^/ предполагается наличие модификатора //m.

Если число *LIMIT* задано и больше нуля, строка расщепляется не более чем на заданное число полей (хотя число полей может быть меньше, если разделителей окажется меньше). Если число *LIMIT* отрицательно, считается, что была задана произвольно большая граница *LIMIT*. Если граница опущена или равна нулю, замыкающие пустые поля удаляются из результата (о чем следует помнить тем, кто будет использовать pop). Если опущен аргумент *EXPR*, расщепляется строка $_. Если шаблон *PATTERN* тоже опущен или задан как пробел, " ", осуществляется расщепление по пробельным символам, /\s+/, после пропуска всех ведущих пробельных символов.

Значение /^/ в аргументе *PATTERN* неявно интерпретируется как /^/m, поскольку использовать его иначе не имеет смысла.

Можно расщеплять строки любой длины:

```
@chars  = split //,   $word;
@fields = split /:/,  $line;
@words  = split " ",  $paragraph,
@lines  = split /^/,  $buffer;
```

Функцию split можно также использовать для расщепления строки на последовательность графем, но гораздо проще использовать для этой цели простое сопоставление с регулярным выражением:

```
@graphs = grep { length } split /(\X)/, $word;
@graphs = $word =~ /\X/g;
```

Шаблон, который может находить соответствие пустой строке или чему-то более длинному, чем пустая строка (например, шаблон, состоящий из одного символа с квантификатором * или ?), расщепляет значение *EXPR* на отдельные символы, если находит пустую строку между символами; непустые соответствия пропускают символы найденного разделителя обычным образом. (Иными словами, шаблон не будет соответствовать в одной точке более одного раза, даже если найдено соответствие нулевой ширины.) Например:

```
print join(":" => split / */, "hi there");
```

выведет "h:i:t:h:e:r:e". Пробел исчезает, поскольку он интерпретируется как часть разделителя. В тривиальном случае пустой шаблон // просто расщепляет

[1] В скалярном контексте split также записывает результат в @_, но это устаревшая особенность.

строку на отдельные символы, и пробелы не исчезают. (При обычном поиске по шаблону // повторяет поиск по последнему успешно найденному шаблону, но в split шаблон представляет собой исключение из этого правила.)

При наличии аргумента *LIMIT* расщепляется только часть строки:

```
my ($login, $passwd, $remainder) = split /:/, $_, 3;
```

Мы рекомендуем такое расщепление в список имен, которое позволит сделать код самодокументирующимся. (При проверке ошибок учитывайте, что значение $remainder окажется неопределенным, если в строке меньше трех полей.) При присваивании списку, когда аргумент *LIMIT* отсутствует, Perl использует для *LIMIT* значение на единицу большее, чем число переменных в списке, чтобы избежать лишней работы. В примере выше *LIMIT* по умолчанию будет иметь значение 4, и $remainder получит только третье поле, а не все оставшиеся поля. В приложениях, где важна скорость выполнения, следует выделять не более полей, чем действительно требуется. (Коварство мощных языков в том, что временами они позволяют делать мощные глупости.)

Ранее мы говорили, что разделители не возвращаются, но если *PATTERN* содержит круглые скобки, в список с результатом включаются подстроки, соответствующие каждой паре скобок, перемежаясь с обычно возвращаемыми полями. Например, вызов:

```
split /([-,])/, "1-10,20";
```

вернет список:

```
(1, "-", 10, ",", 20)
```

Если скобок больше, поле возвращается для каждой пары, даже если каким-то парам не найдено соответствия, и тогда в этих позициях возвращаются неопределенные значения. Поэтому, если сказать:

```
split /(-)|(,)/, "1-10,20";
```

будет получено значение:

```
(1, "-", undef, 10, undef, ",", 20)
```

Аргумент */PATTERN/* можно заменить выражением, которое будет задавать шаблоны, изменяющиеся во время выполнения программы.

В частном случае, когда выражение *EXPR* состоит из одного пробела (" "), функция производит расщепление по пробельным символам, как split без аргументов. Поэтому с помощью split(" ") можно эмулировать режим работы *awk* по умолчанию. Напротив, split(/ /) даст столько нулевых полей в начале, сколько имеется ведущих пробелов. (Другой частный случай, когда вместо регулярного выражения передается строка: она все равно будет интерпретирована как регулярное выражение.) Это свойство можно использовать для удаления из строки ведущих и замыкающих пробелов и замены встречающихся в середине серий пробельных символов на один пробел:

```
$string = join(" ", split(" ", $string));
```

В следующем примере заголовок сообщения RFC 822 расщепляется в хеш, содержащий поля $head{Date}, $head{Subject} и т.д. Здесь используется прием присваи-

вания хешу списка пар, основанный на том обстоятельстве, что разделители чередуются с разделяемыми полями. Чтобы вернуть часть каждого разделителя как часть возвращаемого списка, используются круглые скобки. Поскольку шаблон split гарантированно возвращает значения парами, так как содержит одну пару круглых скобок, операция присваивания хешу гарантированно получит список пар ключ/значение, где каждый ключ является именем поля заголовка. (К сожалению, этот прием приводит к потере информации, если несколько строк имеют одинаковое ключевое поле, например строки Received-By. Увы...)

```
$header =~ s/\n\s+/ /g;      # Слить строки продолжения
%head = ("FRONTSTUFF", split /^(\S*?):\s*/m, $header);
```

В следующем примере обрабатываются записи из файла *passwd*(5) UNIX. Можно опустить chomp, и тогда последним символом $shell будет символ перевода строки.

```
open PASSWD, "/etc/passwd";
while (<PASSWD>) {
    chomp;        # удалить замыкающий перевод строки
    ($login, $passwd, $uid, $gid, $gcos, $home, $shell) =
            split /:/;
    ...
}
```

Вот как можно обработать каждое слово каждой строки каждого входного файла, чтобы создать хеш с частотами слов:

```
while (<>) {
    for my $word (split) {
        $count{$word}++;
    }
}
```

Действие, обратное split, осуществляет join (если не считать, что join может устанавливать между всеми полями только один разделитель). Для разбиения на части строки с полями фиксированной длины используйте unpack.

sprintf

```
sprintf FORMAT, LIST
```

Возвращает строку, отформатированную в соответствии с обычными соглашениями printf, т.е. соглашениями функции *sprintf* библиотеки С. Разъяснение общих принципов читайте в описании *sprintf* (3) или *printf* (3) для вашей системы. Строка *FORMAT* содержит текст со спецификаторами полей, замещаемыми элементами из *LIST*, по одному на каждое поле. Описание полей вы найдете в разделе «Форматы строк» главы 26.

sqrt




```
sqrt EXPR
sqrt
```

Возвращает квадратный корень из *EXPR*. Другие корни, например кубические, можно получить при помощи оператора **, возведя число в дробную степень. Не пытайтесь применять эти подходы к отрицательным числам, поскольку возведе-

ние в степень отрицательных чисел – несколько более сложная задача (и в результате возникает исключение). Но существует стандартный модуль, который умеет делать даже это:

```
use Math::Complex,
print sqrt(-2);        # выведет 1.4142135623731i
```

srand

```
srand EXPR
srand
```

Устанавливает начальное случайное число для оператора rand. Если выражение *EXPR* опущено, используется полуслучайное значение, предоставляемое ядром (если оно поддерживает устройство */dev/urandom*) или полученное, среди прочего, на основе идентификатора процесса и текущего времени. В любом случае, начиная с версии v5.14, функция возвращает начальное число последовательности псевдослучайных чисел (seed). Как правило, нет необходимости вызывать srand, поскольку, если не вызвать ее явно, она будет вызвана неявно при первом использовании оператора rand. Однако в Perl до версии 5.004 это было не так, поэтому, если сценарий должен работать со старыми версиями Perl, он должен явно вызывать srand.

Часто вызываемые программы (такие, как сценарии CGI), которые в качестве начального числа берут просто time ^ $$, могут пасть жертвой того математического свойства, что a^b == (a+1)^(b+1) в одной трети случаев. Поэтому не делайте так. Лучше используйте:

```
srand( time() ^ ($$ + ($$ << 15)) );
```

Для криптографических задач понадобится что-нибудь гораздо более случайное, чем начальное число по умолчанию. В некоторых системах годится устройство */dev/random*. В иных случаях часто применяется контрольная сумма сжатого вывода одной или более программ, возвращающих быстро меняющееся состояние операционной системы. Например:

```
srand (time ^ $$ ^ unpack "%32L*", `ps wwaxl | gzip`);
```

Если вы этим особенно озабочены, ознакомьтесь с модулем Math::TrulyRandom в CPAN.

Не следует вызывать в программе srand несколько раз, кроме, конечно, тех случаев, когда вы точно знаете, что и зачем делаете. Назначение этой функции – дать начальное значение для функции rand, чтобы она могла создавать неодинаковые случайные последовательности при каждом запуске программы. Сделайте это один раз в начале программы, иначе rand не даст вам случайные числа!

stat




```
stat FILEHANDLE
stat DIRHANDLE
stat EXPR
stat
```

В скалярном контексте возвращает логическое значение – признак успешности вызова. В списочном контексте возвращает список из 13 элементов, содержащий

статистику файла, открытого через дескриптор *FILEHANDLE* или *DIRHANDLE*, или файла, имя которого указано в *EXPR*. Обычно используется так:

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
    $atime,$mtime,$ctime,$blksize,$blocks)
            = stat $filename;
```

Не все файловые системы поддерживают полный перечень полей; в неподдерживаемых полях возвращается 0. Значения полей приведены в табл. 27.6.

Таблица 27.6. Значения, возвращаемые stat

Номер	Поле	Значение
0	$dev	Номер устройства файловой системы
1	$ino	Номер inode
2	$mode	Режим файла (тип и разрешения)
3	$nlink	Число (жестких) ссылок на файл
4	$uid	Числовой идентификатор пользователя-владельца файла
5	$gid	Числовой идентификатор группы-владельца файла
6	$rdev	Идентификатор устройства (только специальные файлы)
7	$size	Суммарный размер файла в байтах
8	$atime	Время последнего обращения к файлу в секундах с начала эпохи
9	$mtime	Время последней модификации файла в секундах с начала эпохи
10	$ctime	Время изменения inode (*не* время создания!) в секундах с начала эпохи
11	$blksize	Предпочтительный размер блока для ввода/вывода файловой системы
12	$blocks	Фактическое число выделенных блоков

Поля $dev **и** $ino **в совокупности уникально идентифицируют файл в системе.** $blksize **и** $blocks **определены, скорее всего, только в файловых системах, производных от BSD. Поле** $blocks **(если определено) измеряется в блоках по 512 байт. Значение** $blocks*512 **может существенно отличаться от** $size **для файлов, содержащих невыделенные блоки, или «дыры», которые не учитываются в** $blocks.

Если в stat **передается специальный дескриптор файла, состоящий из символа подчеркивания, фактически *stat*(2) не выполняется, а возвращается текущее содержимое структуры** stat**, полученной в результате последнего вызова** stat, lstat **или оператора проверки файла на основе** stat **(например,** -r, -w **или** -x**).**

Поскольку режим файла содержит его тип и права доступа, чтобы увидеть действующие права, нужно замаскировать часть, содержащую тип файла, и вывести фактические права при помощи printf **или** sprintf **со спецификатором** "%o"**:**

```
$mode = (stat($filename))[2];
printf "Permissions are %04o\n", $mode & 07777;
```

Модуль File::stat **предоставляет удобный механизм доступа по именам:**

```
use File::stat;
$sb = stat($filename);
printf "File is %s, size is %s, perm %04o, mtime %s\n",
```

```
    $filename, $sb->size, $sb->mode & 07777,
    scalar localtime $sb->mtime;
```

Можно также импортировать символические определения различных разрядов режима из модуля Fcntl.

```
use Fcntl ':mode';

$mode = (stat($filename))[2];

$user_rwx      = ($mode & S_IRWXU) >> 6;
$group_read    = ($mode & S_IRGRP) >> 3;
$other_execute =  $mode & S_IXOTH;

printf "Права доступа: %04o\n", S_IMODE($mode), "\n";

$is_setuid     = $mode & S_ISUID;
$is_directory  = S_ISDIR($mode);
```

Две последние инструкции можно было бы заменить операторами -u и -d. Дополнительную информацию вы найдете в описании *stat*(2).

Совет: чтобы получить только размер файла, используйте оператор проверки файла -s, непосредственно возвращающий размер в байтах. Существуют также проверки, возвращающие возраст файла в днях.

state

```
state EXPR
state TYPE EXPR
state EXPR : ATTRS
state TYPE EXPR : ATTRS
```

Объявление state создает переменную в лексической области, по аналогии с объявлением my. Однако содержимое переменных, объявленных как state, сохраняется между вызовами одной и той же подпрограммы; такие переменные инициализируются только один раз – при первом входе в область их видимости, и никогда не инициализируются повторно, в отличие от лексических переменных, инициализирующихся каждый раз при входе в область видимости.

Когда образуется замыкание, оно считается новой подпрограммой, поэтому любые state-переменные инициализируются в новой копии замыкания при первом вызове. Эти переменные не являются статическими в понимании языка C, если только сама подпрограмма не является статической.

Объявление state может использоваться только в области действия прагмы use feature "state". См. раздел «feature» в главе 29. В настоящее время полноценная инициализация state-переменных поддерживается только для скаляров, однако всегда можно использовать скалярную ссылку на массив или хеш.

study




```
study SCALAR
study
```

Эта функция тратит время на изучение *SCALAR*, предполагая многократное сопоставление с шаблоном, прежде чем строка будет модифицирована. При этом мож-

но сберечь, а можно и не сберечь время, что зависит от природы и числа шаблонов поиска и от распределения частот символов в строке, в которой осуществляется поиск, – возможно, придется сравнить время выполнения с применением study и без оного, чтобы узнать, какой вариант быстрее. Больше всего выигрывают от применения study циклы, в которых сканируется множество коротких неменяющихся строк (в том числе постоянные части более сложных шаблонов). Если необходимо найти постоянные строки, привязанные к началу, study вообще не поможет, поскольку никакого сканирования не производится. Одновременно можно иметь только одну активную study; если начать изучение нового скаляра, результаты для первого будут потеряны.

Работает study так. Создается связанный список для каждого символа строки, в которой будет производиться поиск, чтобы знать, например, где располагаются все символы «k». В каждой строке поиска на основании некоторых статических таблиц частот, построенных с помощью C-программ и английского текста, выбирается символ, встречающийся реже всего. Исследуются только те места, где этот символ находится.

Вот, например, цикл, который вставляет элементы, порождающие именной указатель[1], перед каждой строкой, содержащей некоторый шаблон:

```
while (<>) {
    study;
    print ".IX foo\n"    if /\bfoo\b/;
    print ".IX bar\n"    if /\bbar\b/;
    print ".IX blurfl\n" if /\bblurfl\b/;
    ...
    print;
}
```

При поиске /\bfoo\b/ будут рассматриваться только те места в $_, которые содержат «f», так как «f» встречается реже, чем «o». Это дает большой выигрыш, за исключением патологических случаев. Остается только невыясненным, больше ли будет сэкономлено времени, чем было потрачено для построения связанного списка.

Если потребуется выполнять поиск в строках, не известных до момента выполнения, можно сконструировать цикл в строке и скормить эту строку функции eval, чтобы избежать постоянной перекомпиляции шаблонов. Если задать в $/ чтение файлов целиком, этот подход может дать превосходные результаты по скорости – и часто будет быстрее, чем применение специализированных программ вроде *fgrep*(1). Следующий код просматривает список файлов (@files) в поисках списка слов (@words) и выводит имена файлов, где были найдены соответствия, без учета регистра символов:

```
$search = "while (<>) { study;";
for my $word (@words) {
    $search .= "++\$seen{\$ARGV} if /\\b$word\\b/i;\n";
}
$search .= "}";
@ARGV = @files;
```

[1] Здесь авторы имеют в виду тематический указатель имен, приводимый в конце книги. – *Прим. перев.*

```
undef $/;                # читать файл целиком
eval $search;            # перехватывает исключение
die $@ if $@;            # в случае неудачи eval
$/ = "\n";               # восстановить обычный символ завершения ввода
for my $file (sort keys(%seen)) {
    say "$file";
}
```

Сейчас, когда у нас есть оператор qr//, сложные eval этапа выполнения, которые мы видели выше, не столь необходимы. То же самое можно сделать иначе:

```
@pats = ();
for my $word (@words) {
    push @pats, qr/\b${word}\b/i;
}
@ARGV = @files;
undef $/;                # вводить каждый файл целиком
while (<>) {
    for $pat (@pats) {
        $seen{$ARGV}++ if /$pat/;
    }
}
$/ = "\n";               # восстановить обычный символ завершения ввода
for my $file (sort keys(%seen)) {
    say "$file";
}
```

sub

Именованные объявления:

sub *NAME PROTO ATTRS*
sub *NAME ATTRS*
sub *NAME PROTO*
sub *NAME*

Именованные определения:

sub *NAME PROTO ATTRS BLOCK*
sub *NAME ATTRS BLOCK*
sub *NAME PROTO BLOCK*
sub *NAME BLOCK*

Неименованные определения:

sub *PROTO ATTRS BLOCK*
sub *ATTRS BLOCK*
sub *PROTO BLOCK*
sub *BLOCK*

Синтаксис объявлений и определений подпрограмм кажется сложным, но на практике он довольно прост. Вот его основа:

sub *NAME PROTO ATTRS BLOCK*

Все четыре поля не обязательны; единственное ограничение состоит в том, что поля, которые присутствуют, должны быть расположены в указанном порядке и что

необходимо использовать хотя бы одно из полей *NAME* и *BLOCK*. Некоторое время мы не будем обращать внимания на *PROTO* и *ATTRS*; они просто изменяют основной синтаксис. А *NAME* и *BLOCK* – важные параметры, с которыми необходимо разобраться:

- Если указан только аргумент *NAME* без *BLOCK*, – это предварительное объявление имени (чтобы позднее вызвать эту подпрограмму, придется в дальнейшем дать определение, содержащее и *NAME*, и *BLOCK*). Именованные объявления полезны, поскольку синтаксический анализатор обращается с именем особым образом, если знает, что это подпрограмма, определенная пользователем. Такую подпрограмму можно вызвать как функцию или как оператор, подобно любой другой встроенной функции. Иногда их называют *опережающими объявлениями* (*forward declarations*).
- Если указаны и *NAME*, и *BLOCK* – это стандартное определение (definition) именованной подпрограммы (а также объявление, если оно не было сделано раньше). Именованные определения полезны, потому что связывают фактическое значение *BLOCK* (тело подпрограммы) с объявлением. Это все, что мы имеем в виду, говоря, что это определение подпрограммы, а не просто объявление ее. Определение, однако, похоже на объявление, поскольку окружающий код не видит его, и оно не возвращает внедряемое значение, по которому можно ссылаться на подпрограмму.
- Если есть *BLOCK* без *NAME* – это безымянное определение, т.е. анонимная подпрограмма. Так как имя отсутствует, это вообще не объявление, а настоящий оператор, возвращающий ссылку на тело анонимной подпрограммы на этапе выполнения. Это очень удобно для работы с кодом как с данными и позволяет раскидывать случайные фрагменты кода, чтобы использовать их как обратные вызовы, а может быть, даже как замыкания, если оператор определения `sub` ссылается на какие-либо внешние лексические переменные. Это значит, что различные вызовы одного и того же оператора `sub` будут вести бухгалтерию, необходимую для поддержания правильной «версии» каждой такой лексической переменной, которая существует в течение жизни замыкания, даже если начальная область видимости лексической переменной разрушена.

В любом из этих трех случаев за *NAME* и/или перед *BLOCK* может следовать аргумент *PROTO* или *ATTRS* (или оба). Прототип – это список символов в круглых скобках, сообщающих анализатору, как обрабатывать аргументы функции. Атрибуты вводятся с помощью двоеточия и снабжают анализатор дополнительной информацией о функции. Вот типичное определение, включающее все четыре поля:

```
sub numstrcmp ($$) : locked {
    my ($a, $b) = @_;
    return $a <=> $b || $a cmp $b;
}
```

Подробные сведения о списках атрибутов и работе с ними приведены в описании директивы `attributes` в главе 29. См. также главу 7 и раздел «Анонимные подпрограммы» главы 8.

substr




```
substr EXPR, OFFSET, LENGTH, REPLACEMENT
substr EXPR, OFFSET, LENGTH
substr EXPR, OFFSET
```

Извлекает подстроку из строки, заданной выражением *EXPR*, и возвращает ее. Подстрока извлекается, начиная со смещения *OFFSET* символов от начала строки. Если *OFFSET* меньше нуля, смещение отсчитывается от конца строки. Если длина *LENGTH* опущена, возвращается все до конца строки. Если аргумент *LENGTH* имеет отрицательное значение, длина подстроки рассчитывается так, чтобы оставить указанное число символов в конце строки. В противном случае *LENGTH* определяет длину извлекаемой подстроки, как вы, наверное, и предполагали.

Обратите внимание, что речь здесь идет о символах (т.е. о кодах), а не о байтах или графемах. Чтобы перейти к работе с байтами, сначала переведите строку в кодировку UTF-8, а затем повторите попытку. Для работы с графемами используйте метод substr из модуля Unicode::GCString, опубликованного в CPAN.

Функция substr может выступать в качестве левостороннего значения (которому можно присвоить значение). В этом случае выражение *EXPR* тоже должно быть левосторонним значением. Если присваиваемое значение короче, чем подстрока, длина строки уменьшится, а если длиннее, длина строки увеличится. Для сохранения прежней длины строки может потребоваться дополнить или усечь присваиваемое значение посредством sprintf или оператора x. При попытке присвоить что-либо невыделенной области за концом строки substr возбуждает исключение.

Поместить перед строкой "Larry" текущее значение $_ можно так:

```
substr($var, 0, 0) = "Larry"
```

А так можно заменить первый символ в $_ на "Moe":

```
substr($var, 0, 1) = "Moe";
```

И наконец, замена последнего символа в $var на "Curly" выполняется следующим образом:

```
substr($var, -1) = "Curly";
```

Вместо использования substr в качестве левостороннего значения можно передать ей строку в четвертом аргументе *REPLACEMENT*. Это позволит заменить часть *EXPR* и получить прежнюю подстроку на этом месте одной операцией, как это делает splice. Следующий пример заменит последний символ в $var на "Curly" и поместит его в $oldstr:

```
$oldstr = substr($var, -1, 1, "Curly");
```

Не обязательно использовать substr как левостороннее значение только в случае присваивания. Следующий пример заменит все пробелы точками, но только в первых 10 символах строки:

```
substr($var, -10) =~ s/ /./g;
```

Обратите внимание, мы продолжаем говорить о символах. Как и повсюду в этой книге, под символами подразумеваются коды, или символы, видимые программисту, а не графемы, или символы, видимые пользователю. Графемы могут состоять из нескольких кодов каждая, что часто и происходит. Модуль Unicode::GCString из архива CPAN предоставляет альтернативные функции, способные заменить стандартные substr, index, pos и многие другие, которые оперируют логическими графемами, а не кодами.

Если вы собираетесь использовать substr вместо регулярных выражений только потому, что, по вашему мнению, substr должна работать быстрее, вы удивитесь.

Зачастую регулярные выражения оказываются быстрее функции substr, даже при работе с полями фиксированной ширины.

symlink



symlink *OLDNAME*, *NEWNAME*

Создает символическую ссылку *NEWNAME* на файл с именем *OLDNAME*. Возвращает истинное значение в случае успеха и ложное в противном случае. В системах, не поддерживающих символические ссылки, возбуждает исключение на этапе выполнения. Чтобы проверить это, используйте eval для перехвата возможной ошибки:

```
$can_symlink = eval { symlink("",""); 1 };
```

Либо обратитесь к модулю Config. Будьте осторожны при передаче относительной символической ссылки, поскольку она будет интерпретироваться относительно местоположения самой символической ссылки, а не относительно текущего рабочего каталога.

См. также описания link и readlink выше в этой главе.

syscall



syscall *LIST*

Выполняет системный вызов (не команду интерпретатора), указанный в первом элементе списка, и передает оставшиеся элементы в качестве аргументов этого вызова. (Доступ ко многим из этих вызовов теперь стал легче благодаря таким модулям, как POSIX.) Возбуждает исключение, если функция *syscall*(2) не реализована.

Аргументы интерпретируются следующим образом: если аргумент представляет собой число, он передается как целое число языка C. Если нет, передается указатель на строку. Программист отвечает за то, чтобы строка имела достаточную длину и могла принять любой результат, который может быть в нее записан; в противном случае его ожидает аварийное завершение. Нельзя использовать строковой литерал (или другую строку, доступную только для чтения) в качестве аргумента syscall, поскольку Perl должен предполагать, что по любому указателю строки можно производить запись. Если целочисленные аргументы не являются литералами и никогда не интерпретировались в числовом контексте, может потребоваться прибавить к ним 0, чтобы они выглядели как числа.

syscall возвращает значение, которое вернул фактический системный вызов. По соглашениям программирования на C, если системный вызов завершается неудачей, syscall возвращает -1 и устанавливает $! (номер ошибки). Некоторые системные вызовы могут возвращать -1 в случае успеха. Правильный способ обработки таких вызовов состоит в присваивании $!=0 перед вызовом и проверке значения $!, если функция syscall вернула -1.

Не ко всем системным вызовам можно обращаться таким способом. Например, Perl поддерживает передачу системным вызовам до 14 аргументов, чего на практике, как правило, достаточно. Однако не все гладко с системными вызовами, возвращающими несколько значений. Например, вызов syscall(&SYS_pipe) возвращает номер файла на читающем конце созданного им канала. Не существует способа извлечь номер файла другого конца. Во избежание такого рода неприятностей можно обратиться к pipe. Чтобы решить проблему раз и навсегда, напишите

несколько XSUB (модулей внешних подпрограмм, диалект C) для непосредственного доступа к системным вызовам. Затем поместите свой новый модуль в CPAN и станьте ужасно знаменитым.

Следующая подпрограмма возвращает текущее время как число с плавающей запятой, а не целое число секунд, возвращаемое time. (Работает только в системах, поддерживающих системный вызов *gettimeofday*(2).)

```
sub finetime() {
    package main;            # для последующего require
    require "syscall.ph";
    # установить размер буфера для двух 32-разрядных long..
    my $tv = pack("LL", ());
    syscall(&SYS_gettimeofday, $tv, undef) >= 0
        || die "gettimeofday: $!";
    my($seconds, $microseconds) = unpack("LL", $tv);
    return $seconds + ($microseconds / 1_000_000);
}
```

Предположим, что Perl не поддерживает системный вызов *setgroups*(2)[1], а ядро поддерживает. Добраться до него можно таким способом:

```
require "syscall.ph";
syscall(&SYS_setgroups, scalar @newgids, pack("i*", @newgids))
        || die "setgroups: $!";
```

Как сказано в инструкциях по установке Perl, если файл *syscall.ph* отсутствует, может понадобиться запустить *h2ph*. В некоторых системах может потребоваться передать функции pack шаблон "II". Еще тревожнее, что syscall предполагает эквивалентность размеров C-типов int, long и char*. Постарайтесь не считать syscall образцом переносимости в миниатюре.

Более строгий подход к задачам определения времени с более высоким разрешением реализован в модуле Time::HiRes из CPAN.

sysopen




```
sysopen FILEHANDLE, FILENAME, MODE, MASK
sysopen FILEHANDLE, FILENAME, MODE
```

Открывает файл с именем *FILENAME* и связывает его с дескриптором файла *FILEHANDLE*. Если *FILEHANDLE* является выражением, его значение используется как имя или ссылка на дескриптор файла. Если *FILEHANDLE* является переменной с неопределенным значением, необходимое значение будет создано автоматически. Возвращает истинное значение, если вызов успешен, и ложное – в противном случае.

Эта функция является прямым интерфейсом к системному вызову операционной системы *open*(2), за которым следует библиотечный вызов *fdopen*(3). По этой причине вам придется здесь немного прикинуться C-программистом. Возможные значения и биты флагов параметра *MODE* доступны через модуль Fcntl. Поскольку в разных системах поддерживаются разные флаги, не рассчитывайте, что все они будут доступны в вашей системе. За подробностями обращайтесь к странице руководства *open*(2) или ее локальному эквиваленту. Тем не менее в любой системе,

---

[1] Хотя через $(поддерживает.

имеющей более-менее стандартную библиотеку С, должны присутствовать флаги, перечисленные в табл. 27.7.

Таблица 27.7. Флаги для sysopen

Флаг	Значение
O_RDONLY	Только для чтения
O_WRONLY	Только для записи
O_RDWR	Чтение и запись
O_CREAT	Создать файл, если он не существует
O_EXCL	Ошибка, если файл уже существует
O_APPEND	Дописывание в конец файла
O_TRUNC	Усечение файла
O_NONBLOCK	Неблокирующий доступ

Однако существует множество других флагов. В табл. 27.8 перечислены некоторые флаги, употребляемые реже.

Таблица 27.8. Флаги для sysopen, употребляемые реже

Флаг	Значение
O_NDELAY	Старый синоним O_NONBLOCK
O_SYNC	Вернуть управление, только когда данные физически будут записаны в устройство. Можно также встретить O_ASYNC, O_DSYNC и O_RSYNC
O_EXLOCK	flock с помощью LOCK_EX (только рекомендательная блокировка)
O_SHLOCK	flock с помощью LOCK_SH (только рекомендательная блокировка)
O_DIRECTORY	Вернуть ошибку, если файл *не* является каталогом
O_NOFOLLOW	Вернуть ошибку, если последний элемент пути является символической ссылкой
O_BINARY	binmode дескриптора для систем Microsoft. Иногда бывает определен O_TEXT, позволяющий включить противоположное поведение.
O_LARGEFILE	В некоторых системах требуется для файлов, размер которых превышает 2 Гбайт
O_NOCTTY	Открытие файла терминала не делает последний управляющим терминалом процесса, если у вас его еще нет. Обычно не требуется.

Флаг O_EXCL *не* служит для блокировки: в данном случае исключительность означает: если файл уже существует, вызов sysopen завершится неудачей.

Если файл *FILENAME* не существует, а *MODE* содержит флаг O_CREAT, sysopen создаст файл, первоначальные права доступа к которому определяются аргументом *MASK* (или значением 0666, если этот аргумент отсутствует), с учетом текущей маски umask процесса. Такое значение по умолчанию разумно: читайте пояснения в описании umask.

Дескрипторы файлов, открываемые с помощью open и sysopen, могут использоваться взаимозаменяемо. Нет необходимости применять sysread сотоварищи

только потому, что файл оказался открытым посредством sysopen, как этому не препятствует и открытие его посредством open. Каждая из функций может делать то, чего не умеет другая. Обычная функция open может открывать каналы, запускать процессы, добавлять фильтры ввода/вывода, дублировать дескрипторы файлов и преобразовывать численные указатели файлов в дескрипторы файлов. Она также игнорирует ведущие и замыкающие пробельные символы в именах файлов и почитает «–» за специальное имя файла. Однако что касается фактического открытия файлов, sysopen умеет делать то же, что и open.

В следующих примерах приведены эквивалентные вызовы обеих функций. Для ясности мы опустили проверки or die $!, но в своих программах всегда проверяйте возвращаемые значения. Мы ограничимся использованием только флагов, доступных практически в любых операционных системах. Следите за значениями, объединяемыми поразрядным оператором | и передаваемыми в качестве аргумента *MODE*.

- **Открыть файл для чтения:**

```
open(FH, "<", $path);
sysopen(FH, $path, O_RDONLY);
```

- **Открыть файл для записи. При наобходимости создать новый файл или усечь старый:**

```
open(FH, ">", $path);
sysopen(FH, $path, O_WRONLY | O_TRUNC | O_CREAT);
```

- **Открыть файл для дописывания. При необходимости создать его:**

```
open(FH, ">>", $path);
sysopen(FH, $path, O_WRONLY | O_APPEND | O_CREAT);
```

- **Открыть существующий файл для обновления:**

```
open(FH, "+<", $path);
sysopen(FH, $path, O_RDWR);
```

А вот что можно делать с помощью sysopen, ***но нельзя*** с помощью обычной open:

- **Открыть и создать для записи файл, который не должен предварительно существовать:**

```
sysopen(FH, $path, O_WRONLY | O_EXCL | O_CREAT);
```

- **Открыть для дописывания файл, который должен уже существовать:**

```
sysopen(FH, $path, O_WRONLY | O_APPEND);
```

- **Открыть для обновления файл, создать при необходимости новый:**

```
sysopen(FH, $path, O_RDWR | O_CREAT);
```

- **Открыть для обновления файл, который не должен предварительно существовать:**

```
sysopen(FH, $path, O_RDWR | O_EXCL | O_CREAT);
```

- **Открыть файл только для записи без блокировки, но не создавать, если он не существует:**

```
sysopen(FH, $path, O_WRONLY | O_NONBLOCK);
```

Модуль IO::File предоставляет группу объектно-ориентированных синонимов (плюс небольшое число новых функций) для открытия файлов. Соответствующие методы из IO::File или IO::Handle можно вызывать с любым дескриптором, созданным посредством open, sysopen, pipe, socket или accept, даже если эти дескрипторы были инициализированы без использования данного модуля. Фактически, теперь Perl загружает эти модули неявно, чтобы обеспечить доступ к этим функциям.

sysread

$! $@ T X ARG X RO

```
sysread FILEHANDLE, SCALAR, LENGTH, OFFSET
sysread FILEHANDLE, SCALAR, LENGTH
```

Пытается прочесть *LENGTH* символов в переменную *SCALAR* из указанного *FILEHANDLE* посредством системного низкоуровневого вызова *read*(2). Возвращает число прочитанных символов, 0 при достижении EOF[1] или undef в случае ошибки. *SCALAR* будет расти или сокращаться в соответствии с количеством прочитанных символов. Смещение *OFFSET*, если задано, указывает, в какое место строки начать помещать символы, поэтому чтение можно производить в середину строки, которой отводится роль буфера. Пример использования *OFFSET* вы найдете в описании функции syswrite. Если аргумент *LENGTH* имеет отрицательное значение или *OFFSET* указывает позицию за пределами строки, возникает исключение.

Если дескриптор файла не включает фильтр ввода/вывода, отвечающий за кодирование символов, коды прочитанных символов не превышают значения 255, т.е. фактически выполняется чтение байтов.

Будьте готовы решать возникающие проблемы (такие, как прерванные системные вызовы), которые обычно обрабатывает стандартная система ввода/вывода. Поскольку sysread выполняет операции в обход стандартной системы ввода/вывода, не смешивайте ее с другими функциями чтения, print, printf, write, seek, tell или eof в работе с одним и тем же файловым дескриптором, если только вы не заинтересованы в сильнодействующем колдовстве (и/или головной боли). Имейте также в виду, что при чтении файла, содержащего символы в кодировке UTF-8, UTF-16 или любой другой многобайтной кодировке, граница буфера может попасть на середину символа. Поэтому лучше заранее установить кодировку и выполнять чтение посимвольно, а не побайтно.

Обратите внимание, что если дескриптор файла помечен как :utf8, вместо байтов будет выполняться чтение символов Юникода (*LENGTH*, *OFFSET* и возвращаемое значение функции sysread будут выражаться в символах Юникода). Фильтр :encoding(...) неявно включает и фильтр :utf8.

sysseek

$! X ARG

```
sysseek FILEHANDLE, POSITION, WHENCE
```

Устанавливает системную позицию *FILEHANDLE* с помощью системного вызова *lseek*(2). Операция выполняется в обход стандартной системы ввода/вывода,

---

[1] Функции syseof нет, и это нормально, поскольку eof в любом случае не очень хорошо работает с файлами устройств (например, с терминалами). Используйте sysread и сравнивайте возвращаемое значение с 0, чтобы определить конец файла.

поэтому, смешивая ее с операциями чтения (отличными от sysread), print, write, seek, tell или eof, вы можете нажить неприятности. *FILEHANDLE* может быть выражением, возвращающим имя дескриптора файла. Возможные значения *WHENCE*: 0 устанавливает новую позицию *POSITION*, 1 устанавливает позицию в *POSITION* относительно текущей позиции, и 2 устанавливает позицию в EOF плюс *POSITION* (обычно отрицательную). Значением *WHENCE* могут быть константы SEEK_SET, SEEK_CUR и SEEK_END из стандартных модулей IO::Seekable, POSIX или Fcntl; константы более переносимы и более удобны.

Возвращает новую позицию в байтах или undef – в случае неудачи. Нулевая позиция возвращается в виде особой строки "0 but true", которая может использоваться как число, не вызывая вывода предупреждений и не требуя использования // die вместо более привычного || die.

Внимание: аргумент *POSITION* определяет значение позиции в байтах, а не в символах, независимо от того, задействован ли в дескрипторе файла какой-либо фильтр ввода/вывода для декодирования. Однако все функции Perl, выполняющие чтение файлов, *используют* тот или иной фильтр кодирования/декодирования, потому есть вероятность прочитать часть «символа» и получить недопустимую строку. Старайтесь не смешивать вызовы sysseek или seek с функциями ввода/вывода при работе с дескрипторами файлов, предусматривающими обработку многобайтных кодировок.

system

$! $? XT

```
system PATHNAME LIST
system LIST
```

Выполняет любую программу в системе и возвращает код ее завершения (вместо вывода). Чтобы перехватить вывод команды, используйте обратные апострофы или qx//. Функция system действует так же, как exec, за исключением того, что system сначала выполняет fork, а затем, после exec, ждет завершения выполняемой программы. Это значит, что она запускает указанную программу и возвращается по ее завершении, тогда как exec *заменяет* выполняемую программу новой и уже не возвращается, если замена прошла успешно.

Обработка аргументов зависит от их количества; алгоритм этой обработки приведен в описании exec, включая определение необходимости вызова интерпретатора команд и попытки обмана программы относительно ее имени указанием отдельного *PATHNAME*.

Поскольку system и обратные апострофы блокируют SIGINT и SIGQUIT, отправка одного из этих сигналов (например, от Control-C) выполняемой программе не прерывает основную программу. Но другая программа, которую вы выполняете, получит сигнал. Проверяйте значение, возвращаемое system, чтобы знать, завершилась ли запущенная вами программа корректно.

```
@args = ("command", "arg1", "arg2"),
system(@args) == 0
    || die "system @args failed: $?"
```

Возвращаемое значение представляет собой код завершения программы, полученный системным вызовом *wait*(2). В традиционной семантике для получения реального кода завершения нужно разделить полученное значение на 256 (или

сдвинуть вправо на 8 разрядов). Дело в том, что в младшем байте есть кое-что еще. (В действительности два кое-чего.) Младшие семь разрядов содержат номер сигнала, прервавшего процесс (если он был получен), а восьмой бит указывает, был ли сохранен дамп памяти процесса. Можно проверить все варианты ошибок, включая сигналы и дамп памяти, исследовав значение переменной $? ($CHILD_ERROR):

```
$exit_value  = $? >> 8;
$signal_num  = $? & 127;   # или 0x7f, или 0177, или 0b0111_1111
$dumped_core = $? & 128;   # или 0x80, или 0200, или 0b1000_0000
```

Если программа запущена через системный интерпретатор команд[1] вследствие того, что единственный аргумент содержит метасимволы интерпретатора, обычные коды возврата подвергаются воздействию специфических особенностей и возможностей этого интерпретатора. Иными словами, в этом случае вы можете оказаться не в состоянии восстановить подробную информацию, как описано ранее.

syswrite




```
syswrite FILEHANDLE, SCALAR, LENGTH, OFFSET
syswrite FILEHANDLE, SCALAR, LENGTH
syswrite FILEHANDLE, SCALAR
```

Пытается записать *LENGTH* байтов из переменной *SCALAR* в указанный *FILEHANDLE* посредством системного вызова *write*(2). Возвращает число записанных байтов или undef – в случае ошибки. Смещение *OFFSET*, если задано, указывает, с какой позиции в строке начать запись. (Это может понадобиться при использовании строки как буфера или если возникнет необходимость восстановления при частичной записи.) Отрицательное смещение *OFFSET* указывает, что запись должна начаться с позиции, отсчитанной от конца строки. Если строка *SCALAR* пустая, допустимо только значение *OFFSET*, равное 0. Если аргумент *LENGTH* имеет отрицательное значение или *OFFSET* указывает позицию вне строки, возникает исключение.

Чтобы скопировать данные из дескриптора файла FROM в дескриптор файла TO, можно сказать:

```
use Errno qw/EINTR/;
$blksize = (stat FROM)[11] || 16384;   # предпочтительный размер блока?
while ($len = sysread FROM, $buf, $blksize) {
    if (!defined $len) {
        next if $! == EINTR;
        die "Системная ошибка чтения: $!";
    }
    $offset = 0;
    while ($len) {                     # Обработка неполной записи.
        $written = syswrite TO, $buf, $len, $offset;
        die "Системная ошибка записи: $!" unless defined $written;
        $offset += $written;
        $len    -= $written;
    }
}
```

[1] По определению это */bin/sh* или соответствующий эквивалент для вашей платформы, но не тот интерпретатор, под управлением которого в данный момент работает пользователь.

Следует подготовиться к решению возможных проблем, например, при частичной записи. Поскольку syswrite выполняет запись в обход стандартной библиотеки ввода/вывода C, не смешивайте ее вызовы с операциями чтения (кроме sysread), записи (например, print, printf или write) или другими функциями стандартного ввода/вывода, такими как seek, tell или eof, если только вы не заинтересованы в сильнодействующем колдовстве.[1]

Обратите внимание, что если для дескриптора файла активен фильтр :utf8, вместо байтов будет выполняться запись символов Юникода в кодировке UTF-8, а значения *LENGTH*, *OFFSET* и возвращаемое значение функции sysread будут выражаться в символах Юникода (в кодировке UTF-8). Фильтр :encoding(...) неявно активизирует фильтр :utf8.

tell

```
tell FILEHANDLE
tell
```

Возвращает текущую позицию в файле (в байтах, отсчет с нуля) для *FILEHANDLE*. Обычно это значение затем передается на вход функции seek, чтобы вернуться к текущей позиции. *FILEHANDLE* может быть выражением, возвращающим имя текущего дескриптора файла, или ссылкой на объект дескриптора файла. Если аргумент *FILEHANDLE* опущен, возвращается позиция в файле, из которого последний раз осуществлялось чтение. Позиции в файле имеют смысл только для обычных файлов. Для устройств, каналов и сокетов позиция в файле не существует.

Обратите внимание на слова «*в байтах*»: даже если дескриптор файла настроен на работу с символами (например, был включен фильтр :encoding(utf8)), tell всегда возвращает смещения в байтах, а не в символах (потому что иначе функции seek и tell выполнялись бы чрезвычайно медленно).

Функции systell не существует. Используйте для этой цели sysseek(FH, 0, 1). См. пример вызова tell в описании seek.

Не используйте tell (или любую другую операцию буферизованного ввода/вывода) с дескрипторами файлов, если к ним применялись функции sysread, syswrite или sysseek. Эти функции игнорируют буферизованный ввод/вывод, а tell – нет.

telldir



```
telldir DIRHANDLE
```

Возвращает текущую позицию после вызова readdir с дескриптором каталога *DIRHANDLE*. Это значение можно передать seekdir для доступа к конкретному месту в каталоге. Для этой функции действуют те же предостережения о возможном сжатии каталога, что и для соответствующей программы системной библиотеки. Функция может быть реализована не везде, где есть readdir. Даже если она реализована, возвращаемое значение нельзя использовать в расчетах. Это не вполне прозрачное значение, имеющее смысл только для seekdir.

[1] Или головной боли.

tie



```
tie VARIABLE, CLASSNAME, LIST
```

Связывает переменную с классом пакета, который обеспечивает реализацию этой переменной. *VARIABLE* – переменная (скаляр, массив или хеш) или typeglob (представляющий дескриптор файла) для связывания. *CLASSNAME* – имя класса, реализующего объекты надлежащего типа.

Любые дополнительные аргументы передаются соответствующему конструктору класса с именем TIESCALAR, TIEARRAY, TIEHASH или TIEHANDLE. (Если требуемый конструктор не найден, возникает исключение.) Обычно это те аргументы, которые могут быть переданы в функцию C *dbm_open*(3), но их значение зависит от пакета. Объект, возвращаемый конструктором, в свою очередь возвращается функцией tie, что может пригодиться для обращения к другим методам в *CLASSNAME*. (Доступ к объекту может также осуществляться через функцию tied.) Поэтому класс для связывания хеша с реализацией ISAM может предоставлять дополнительный метод для последовательного обхода множества ключей («S» в аббревиатуре ISAM означает «sequential» – «последовательный»), поскольку обычная реализация DBM может этого не делать.

Такие функции, как keys и values, могут возвращать огромные списки, если используются с такими большими объектами, как файлы DBM. Для их обхода предпочтительнее использовать функцию each. Например:

```
use NDBM_File;
tie(%ALIASES, "NDBM_File", "/etc/aliases", 1, 0)
    || die "Невозможно открыть псевдонимы: $!";
while (($key,$val) = each %ALIASES) {
    say "$key = $val";
}
untie %ALIASES;
```

Класс, реализующий хеш, должен предоставлять следующие методы:

```
TIEHASH CLASS, LIST
FETCH SELF, KEY
STORE SELF, KEY, VALUE
DELETE SELF, KEY
CLEAR SELF
EXISTS SELF, KEY
FIRSTKEY SELF
NEXTKEY SELF, LASTKEY
SCALAR SELF
DESTROY SELF
UNTIE SELF
```

Класс, реализующий обычный массив, должен предоставлять следующие методы:

```
TIEARRAY CLASS, LIST
FETCH SELF, KEY
STORE SELF, KEY, VALUE
FETCHSIZE SELF
STORESIZE SELF, COUNT
CLEAR SELF
PUSH SELF, LIST
```

```
POP SELF
SHIFT SELF
UNSHIFT SELF, LIST
SPLICE SELF, OFFSET, LENGTH, LIST
EXTEND SELF, COUNT
DESTROY SELF
UNTIE SELF
```

Класс, реализующий скаляр, должен предоставлять следующие методы:

```
TIESCALAR CLASS, LIST
FETCH SELF,
STORE SELF, VALUE
DESTROY SELF
UNTIE SELF
```

Класс, реализующий дескриптор файла, должен предоставлять следующие методы:

```
TIEHANDLE CLASS, LIST
READ SELF, SCALAR, LENGTH, OFFSET
READLINE SELF
GETC SELF
WRITE SELF, SCALAR, LENGTH, OFFSET
PRINT SELF, LIST
PRINTF SELF, FORMAT, LIST
BINMODE SELF
EOF SELF
FILENO SELF
SEEK SELF, POSITION, WHENCE
TELL SELF
OPEN SELF, MODE, LIST
CLOSE SELF
DESTROY SELF
UNTIE SELF
```

Не все перечисленные выше методы требуется реализовывать: модули Tie::Hash, Tie::Array, Tie::Scalar и Tie::Handle предоставляют базовые классы с приемлемыми методами по умолчанию. Подробное обсуждение этих методов вы найдете в главе 14. В отличие от dbmopen, функция tie не станет загружать модуль через use или require – программист должен сделать это сам, явно. Интересные реализации tie можно найти в модулях DB_File и Config.

tied

```
tied VARIABLE
```

Возвращает ссылку на объект, лежащий в основе скаляра, массива, хеша или typeglob, содержащихся в *VARIABLE* (то же значение, которое первоначально возвращалось вызовом tie, связавшим переменную с пакетом). Возвращает неопределенное значение, если *VARIABLE* не привязана к пакету. Поэтому, например, можно использовать:

```
ref tied %hash
```

чтобы определить пакет привязки вашего хеша. (Если вы забыли.)

time

```
time
```

Возвращает число секунд без учета високосных лет после «начала эпохи», обычно это 00:00:00 1 января 1970 года, UTC (Universal Coordinated Time).[1] Возвращаемое значение можно передать функции utime, а также функциям gmtime и localtime для сравнения с временем модификации файла и доступа к файлу, полученных от stat.

```
$start = time();
system("некая медленная команда");
$end   = time();
if ($end - $start > 1) {
    say "Программа запущена:    ", scalar localtime($start);
    say "Программа завершилась. ", scalar localtime($end);
}
```

Для измерения интервалов времени с более высокой точностью, чем целое число секунд, используйте модуль Time::HiRes, входящий в состав дистрибутива Perl начиная с версии v5.8 и доступный в CPAN для более ранних версий.

times



```
times
```

В списочном контексте возвращает список из четырех элементов, которые дают время в секундах (возможно, дробное) работы CPU в пространстве пользователя и в пространстве ядра, для данного процесса и его завершившихся порожденных процессов.

```
($user, $system, $cuser, $csystem) = times();
printf "Данный pid и его потомки использовали %.3f секунд\n",
            $user + $system + $cuser + $csystem;
```

В скалярном контексте возвращает только время работы в пространстве пользователя. Например, для измерения времени выполнения фрагмента кода Perl:

```
$start = times();
...
$end = times();
printf "Это заняло %.2f секунд в пространстве пользователя\n",
       $end - $start;
```

tr///



```
tr///
y///
```

Это оператор транслитерации (иногда по ошибке его называют оператором трансляции), подобный оператору y/// в программе UNIX *sed*, только лучше, по всеобщему скромному мнению. См. главу 5.

---

[1] Не путать с «эпопеей», что относится к созданию UNIX. (У других операционных систем могут быть другие эпохи, не говоря уже о других эпопеях.)

Чтобы работать со значением, доступным только для чтения, не опасаясь исключений, используйте модификатор /r, впервые появившийся в версии v5.14.

```
say "bookkeeper" =~ tr/boep/peob/r; # выведет "peekkoobor"
```

truncate

$! X ARG X T X U

```
truncate FILEHANDLE, LENGTH
truncate EXPR, LENGTH
```

Усекает файл, открытый в *FILEHANDLE* или заданный именем в *EXPR*, до указанной длины. Возбуждает исключение, если *ftruncate*(2) или его эквивалент не реализованы в данной системе. (Усечение файла всегда можно произвести, скопировав его начало, при наличии достаточного дискового пространства.) Возвращает истинное значение в случае успеха и undef в противном случае.

Поведение функции не определено, если значение *LENGTH* окажется больше текущей длины файла. Однако в традиционных файловых системах для UNIX длина файла увеличивается до указанного значения, а «несуществующие данные» возвращаются ядром в виде нулевых байтов.

Позиция чтения/записи в файле *FILEHANDLE* не изменяется. После усечения файла может потребоваться вызвать функцию seek перед записью.

uc

$_ T

```
uc EXPR
uc
```

Возвращает версию *EXPR* в верхнем регистре. Это внутренняя функция, реализующая escape-последовательность \U в интерполируемых строках. Для перевода в заглавный регистр используйте функцию ucfirst.

Не используйте uc для реализации сравнения без учета регистра символов, как, возможно, вы делали это при работе со строками ASCII, потому что для строк Юникода этот прием дает неверный результат. Для этих целей лучше либо использовать функцию fc (выполняющую свертку регистра) из модуля Unicode::CaseFold в CPAN, либо задействовать прагму use feature "fc" в Perl версии v5.16 и более поздних. Дополнительные сведения можно найти в разделе «Ошибочные представления о регистре» главы 6.

Функция uc игнорирует коды символов в диапазоне 128–256, если к строке не применяется семантика Юникода (и не действует режим национальных настроек), о чем трудно догадаться. Особенность unicode_strings гарантирует включение семантики Юникода даже для этих кодов. См. главу 6.

ucfirst

$_ T

```
ucfirst EXPR
ucfirst
```

Возвращает версию *EXPR*, в которой первые символы слов переведены в *заглавный регистр*, а остальные символы не изменяются. *Заглавный регистр* – это регистр начальной прописной буквы, за которой следуют (предположительно) буквы в нижнем регистре. Такой регистр используется, например, в первом слове пред-

ложения, в имени человека, в газетном заголовке, а также в большинстве слов названия книги[1]. Символы, не имеющие заглавного регистра, преобразуются в верхний регистр. Это внутренняя функция, реализующая escape-последовательность \u в строках, заключенных в двойные кавычки.

Например, если поместить символ U+FB02 LATIN SMALL LIGATURE FL в начало слова «flower» (т.е. "\x{FB02}ower") и поставить это слово первым в предложении, после перевода в заглавный регистр будет получено слово «Flower», а не «FLower». Однако в результате перевода в верхний регистр будет получено слово «FLOWER».

Перевести первый символ в заглавный регистр, а все остальное – в нижний можно так:

```
ucfirst(substr($word, 0, 1))   lc(substr($word, 1))
```

Не используйте

```
ucfirst lc $word
```

(разве что вам симпатичен культурный империализм) или "\u\L$word", потому что для некоторых символов может быть получен неверный результат. Результат перевода в заглавный регистр символов, переведенных ранее в нижний регистр, не всегда совпадает с результатом перевода в заглавный регистр исходных символов.

Поскольку заглавный регистр имеет смысл только для первого символа в строке, за которым следуют символы нижнего регистра, мы не видим каких-либо причин, оправдывающих перевод в заглавный регистр *всех* символов в строке. И все же покажем, как это сделать:

```
$string =~ s/ ( (?= \p{CWT} ) \X ) /\u$1/gx;
```

Свойство CWT, использованное здесь, имеет полное имя Changes_When_Titlecased=True, но его слишком долго вводить с клавиатуры, поэтому и было выбрано официальное сокращение.

См. в описании uc замечания, касающиеся особенности unicode_strings.

umask



```
umask EXPR
umask
```

Устанавливает пользовательскую маску режима доступа создаваемых процессом файлов и возвращает ее прежнее значение, используя системный вызов *umask*(2). Маска сообщает операционной системе, какие биты прав доступа *отключать* при создании новых файлов, в том числе каталогов. Если аргумент *EXPR* опущен, функция просто возвращает текущую маску. Например, чтобы разрешить биты прав доступа для «владельца» и сбросить биты прав доступа для «других», попробуйте что-нибудь вроде:

```
umask((umask() & 077) | 7); # не изменять биты группы
```

Помните, что маска представляет собой число, обычно задаваемое в восьмеричном виде, а не строку восьмеричных цифр. Если же у вас есть только строка, об-

[1] Имеется в виду англоязычная традиция, где каждое слово в названии книги, кроме артиклей и союзов, принято начинать с заглавной буквы. – *Прим. перев.*

ратитесь к функции oct. Помните также, что биты маски являются дополнениями для обычных разрешений.

Права доступа UNIX rwxr-x--- представляются как три группы по три бита или три восьмеричные цифры: 0750 (ведущий 0 указывает, что это восьмеричное число, и не считается за цифру). Поскольку биты прав доступа в маске перевернуты, она представляет выключенные биты разрешений. Значения прав доступа (или «режима»), передаваемые mkdir или sysopen, модифицируются маской пользователя, поэтому даже если потребовать от sysopen создать файл с режимом 0777, при маске пользователя 0022, файл будет создан с правами доступа 0755. Если маска равна 0027 (группа не может писать; остальные не могут читать, писать или выполнять), вызов sysopen с маской *MASK*, равной 0666, создаст файл с режимом 0640 (поскольку 0666 & ~0027 равно 0640).

Вот некоторые советы: указывайте режим создания 0666 для обычных файлов (в sysopen) и 0777 для исполняемых файлов и каталогов (в mkdir). Это дает пользователям свободу выбора: если им нужны защищенные файлы, они выбирают маску процесса 022, 027 или даже особенно асоциальную маску 077. Программам навряд ли следует возлагать на пользователя принятие решения относительно политики безопасности. Исключением из этого правила служат программы, осуществляющие запись в файлы, которые должны быть закрытыми: почтовые файлы, «куки»-файлы веб-браузера, файлы *.rhosts* и т.д.

Если системный вызов *umask*(2) не реализован в системе, при попытке ограничить свой собственный доступ (т.е. (*EXPR* & 0700) > 0) на этапе выполнения возникнет исключение. Если вызов *umask*(2) не реализован, и вы не пытаетесь ограничить свой собственный доступ, функция просто вернет undef.

undef

X RO

```
undef EXPR
undef
```

undef – это имя, которым мы обозначаем абстракцию, известную как «неопределенное значение». По удачному стечению обстоятельств оно также является именем функции, которая всегда возвращает неопределенное значение. Мы благополучно смешиваем одно с другим[1].

По некоторому совпадению функция undef может также явно сделать неопределенным объект, если передать его имя в качестве аргумента. Аргумент *EXPR*, если присутствует, должен быть левосторонним значением. Поэтому применять функцию можно только к скалярной величине, массиву или хешу целиком, имени подпрограммы (с префиксом &) или typeglob. Память, занимаемая объектом, освобождается, и может повторно использоваться (хотя в большинстве операционных систем она не возвращается в систему). Для большинства специальных переменных действие функции undef, вероятно, обманет ваши ожидания. Передача этой функции переменной, доступной только для чтения, такой как $1, возбуждает исключение.

---

[1] С другой стороны, Perl 6 благополучно наводит порядок в undef и смешивает другие понятия.

Функция undef **является унарным оператором, а не списочным, поэтому сделать неопределенным можно только одно значение за раз. Вот несколько примеров использования** undef **как унарного оператора:**

```
undef $foo;
undef $bar{"blurfl"};  # Не то же, что delete $bar{"blurfl"};
undef @ary;
undef %hash;
undef &mysub;
undef *xyz;            # разрушит $xyz, @xyz, %xyz, &xyz и так далее.
```

undef **без аргумента используется как простое значение:**

```
select(undef, undef, undef, $naptime);

return (wantarray ? () : undef) if $they_blew_it;
return if $they_blew_it; # то же самое
```

undef **может выступать в качестве заполнителя в левой части присваивания списку, тогда соответствующее значение в правой части просто отбрасывается. В другом качестве использовать** undef **как левостороннее значение нельзя.**

```
($a, $b, undef, $c) = &foo; # Игнорировать третье возвращаемое значение
```

Кроме того, не пытайтесь сравнивать что-либо с undef **– получится не то, что вы думаете. Сравнение будет выполнено с** 0 **или пустой строкой. Выяснить, определено ли значение, можно посредством функции** defined **или оператора //.**

unlink

$_ $! XT

```
unlink LIST
unlink
```

Удаляет файлы, перечисленные в списке[1]. Возвращает число удаленных файлов. Например:

```
$count = unlink("a", "b", "c");
unlink @goners;
unlink glob("*.orig");
```

Функция unlink **не станет удалять каталоги, если вы не являетесь суперпользователем и Perl запущен без ключа командной строки *-U*. Помните, что даже если оба эти условия выполнены, удаление каталога таким способом может нанести Серьезный Ущерб файловой системе. Используйте для этих целей** rmdir.

Вот простая команда *rm* с очень простым контролем ошибок:

```
#!/usr/bin/perl
@cannot = grep {not unlink} @ARGV;
die "$0: невозможно выполнить unlink @cannot" if @cannot;
```

---

[1] Фактически в файловой системе POSIX она удаляет записи из каталога (имена файлов), которые ссылаются на реальные файлы. Поскольку ссылка на файл может существовать в нескольких каталогах, файл будет удален только после удаления последней ссылки на этот файл.

unpack



```
unpack TEMPLATE EXPR
```

Оказывает действие, обратное pack: распаковывает строку (*EXPR*), представляющую структуру данных, в список значений согласно шаблону *TEMPLATE* и возвращает эти значения. Шаблоны функций pack и unpack описываются в главе 26.

unshift



```
unshift ARRAY, LIST
```

Выполняет действие, противоположное shift. (Или противоположное push, в зависимости от того, как на это посмотреть.) Добавляет список *LIST* в начало массива и возвращает новое число элементов в массиве:

```
unshift(@ARGV, "-e", $cmd) unless $ARGV[0] =~ /^-/;
```

Обратите внимание, что *LIST* добавляется целиком, а не по одному элементу, поэтому добавляемые элементы сохраняют свой порядок. Для обращения порядка следования элементов используйте reverse.

Начиная с версии v5.14, unshift может принимать ссылку на «неосвященный» хеш или массив, которая будет разыменована автоматически. Эта особенность unshift считается экспериментальной. Ее поведение может измениться в будущем.

untie

```
untie VARIABLE
```

Разрывает связь между переменной или typeglob в *VARIABLE*, и пакетом, к которому она привязана. См. tie и всю главу 14, особенно раздел «Неочевидная ловушка при отвязывании».

use



```
use MODULE VERSION LIST
use MODULE VERSION ()
use MODULE VERSION
use MODULE LIST
use MODULE ()
use MODULE
use VERSION
```

Объявление use загружает указанный модуль, если он не был загружен ранее, а затем импортирует из него подпрограммы и переменные в текущий пакет. (Говоря формальным языком, в текущий пакет импортируется некоторая семантика из указанного модуля, обычно за счет создания псевдонимов для некоторых подпрограмм и переменных.) Большинство объявлений use выглядит так:

```
use MODULE LIST;
```

Это в точности эквивалентно высказыванию:

```
BEGIN { require MODULE; import MODULE LIST }
```

BEGIN заставляет require и import выполняться во время компиляции. require обеспечивает загрузку модуля в память, если она еще не произведена. import не является встроенной функцией – это обычный вызов метода класса в пакете с именем *MODULE*, сообщающий этому модулю о необходимости загрузить список особенностей в текущий пакет. Модуль может реализовать свой метод импорта любым способом, хотя большинство модулей получают его путем наследования от класса Exporter, определенного в модуле Exporter. Дополнительные сведения см. в главе 11, а также в модуле Exporter. Если метод import не найден, вызов просто пропускается.

Чтобы предотвратить изменение своего пространства имен, можно явно передать пустой список:

```
use MODULE ();
```

Это в точности эквивалентно следующему коду:

```
BEGIN { require MODULE }
```

Если первый аргумент use является номером версии, например v5.12.3, выполняемая в данный момент версия Perl должна быть не ниже указанной. Если текущая версия Perl меньше *VERSION*, выводится сообщение об ошибке, и выполнение Perl немедленно завершается. Это полезно для проверки текущей версии Perl перед загрузкой библиотечных модулей, зависящих от более новых версий, поскольку иногда нам приходится «ломать» ошибочные функции старых версий Perl. (Мы стараемся ломать не больше, чем необходимо. На самом деле, мы часто стараемся ломать меньше, чем необходимо.)

Подтверждением слов «стараемся ломать меньше» может служить сохранившаяся поддержка старых номеров версий в формате:

```
use 5.005_03:
```

Однако для лучшего согласования с промышленными стандартами все версии Perl теперь принимают (а мы предпочитаем видеть) формат из трех компонентов:

```
use 5.12.0;  # Версия 5, подверсия 12, уровень исправлений 0.
use v5.12.0; # то же самое
use v5.12;   # то же самое, но не забудьте добавить символ "v"!
use 5.012;   # то же самое, для совместимости со старыми версиями perl
use 5.12;    # НЕВЕРНО!
```

Если за аргументом *MODULE* следует аргумент *VERSION*, use вызовет метод VERSION в классе *MODULE* с указанным значением *VERSION* в качестве аргумента. Обратите внимание на отсутствие запятой после *VERSION*! Метод VERSION по умолчанию, наследуемый от класса UNIVERSAL, возбуждает исключение (croak), если указанная версия превышает значение переменной $Module::VERSION.

Кроме того, начиная с версии v5.10, директива use VERSION также загружает прагму feature и включает все возможности, доступные в запрошенной версии. См. раздел «feature» в главе 29. Аналогично Perl v5.12 и более поздних версий накладывает дополнительные ограничения в лексической области, как при использовании use strict (за исключением того, что файл *strict.pm* в действительности не загружается).

Поскольку use обеспечивает широко открытый интерфейс, прагмы (директивы компилятора) тоже реализуются посредством модулей. Вот примеры реализованных в настоящее время прагм:

```
use autouse "Carp" => qw(carp croak);
use bignum;
use constant PI => 4 * atan2(1,1);
use diagnostics;
use integer;
use lib "/opt/projects/spectre/lib";
use locale;
use sigtrap qw(die INT QUIT);
use sort qw(stable _quicksort _mergesort);
use strict qw(subs vars refs);
use threads;
use warnings qw(numeric uninitialized);
use warnings qw(FATAL all);
```

Многие из таких модулей прагм импортируют семантику в текущую область лексической видимости. (Этим они отличаются от обычных модулей, импортирующих символы только в текущий пакет, почти не имеющий отношения к текущей лексической области видимости – помимо того, что текущая лексическая область видимости компилируется с учетом этого пакета. Это значит... впрочем, не имеет значения, читайте главу 11.)

Поскольку объявление use действует на этапе компиляции, оно не может использоваться в условных инструкциях, управляющих потоком выполнения программы. В частности, если поместить use в ветку else условной инструкции, это не предотвратит ее обработку компилятором. Если модули или прагмы должны загружаться только при определенных условиях, реализовать это можно посредством прагмы if:

```
use if $] < 5.008, "utf8";
use if WANT_WARNINGS, warnings => qw(all);
```

Имеется соответствующее объявление no, которое «деимпортирует» значения, первоначально импортированные use и ставшие ненужными:

```
no integer;
no strict qw(refs);
no warnings qw(deprecated);
```

Особую осторожность следует проявлять при использовании формы no *VERSION*. Она может использоваться *только* для проверки того, что действующий интерпретатор Perl имеет более раннюю версию, чем аргумент *VERSION*, а не для отмены побочного эффекта use VERSION, выражающегося в активизации возможностей языка.

См. список стандартных директив в главе 29.

utime



```
utime LIST
```

Изменяет время обращения и время модификации для всех перечисленных файлов. Первые два элемента списка должны задавать время обращения и модификации в *числовом* виде, причем именно в этом порядке. Функция возвращает число успешно измененных файлов. Время изменения каждого файла в его индексном узле (inode) устанавливается равным текущему времени. Вот пример команды *touch*, устанавливающей дату модификации файла (в предположении, что вы его владелец) примерно на месяц вперед:

```
#!/usr/bin/perl
# montouch - датирует файлы более поздним числом: сейчас + 1 месяц
$day = 24 * 60 * 60;          # 24 часа в виде секунд
$later = time() + 30 * $day; # 30 дней - это примерно месяц
utime $later, $later, @ARGV;
```

а вот более сложная команда в стиле *touch* с некоторой проверкой ошибок:

```
#!/usr/bin/perl
# montouch - датирует файлы более поздним числом: сейчас + 1 месяц
$later = time() + 30 * 24 * 60 * 60;
@cannot = grep {not utime $later, $later, $_} @ARGV;
die "$0: невозможно выполнить touch @cannot." if @cannot;
```

Чтобы прочесть отметки времени существующих файлов, используйте stat, а затем пропустите соответствующие поля через localtime или gmtime перед выводом.

В сетевой файловой системе NFS будет использоваться время на сервере NFS, а не время на локальном компьютере. Если есть проблемы синхронизации времени, время будет отличаться на сервере NFS и локальном компьютере. Команда *touch*(1) в UNIX обычно использует именно такую форму, а не ту, что показана в первом примере.

Если передать в одном из первых двух аргументов значение undef, это будет эквивалентно передаче значения 0, и результат будет иной, чем если передать undef в обоих аргументах. Это также вызовет предупреждение о неинициализированном аргументе.

В системах, поддерживающих *futimes*(2), наряду с именами файлов в списке можно передавать дескрипторы файлов. В системах, не поддерживающих *futimes*(2), передача дескриптора файла вызывает исключение. Чтобы функция успешно распознала дескрипторы файлов, они должны передаваться как typeglobs или ссылки на typeglobs; голые слова будут интерпретироваться как имена файлов.

```
utime($then, $then, *SOME_HANDLE);
```

values





```
values HASH
values ARRAY
```

Возвращает список всех значений в хеше *HASH* или в массиве *ARRAY*. Значения возвращаются в порядке, кажущемся случайным, но это тот же порядок, который создают функции keys и each для того же хеша. Как ни странно, для сортировки хеша по значениям обычно требуется функция keys, поэтому за примером обратитесь к описанию keys.

С помощью этой функции можно модифицировать значения в хеше, потому что возвращаемый список содержит не копии значений, а их псевдонимы. (В ранних версиях для этого требовался срез хеша.)

```
for (@hash{keys %hash}) { s/foo/bar/g }     # по-старому
for (values %hash)      { s/foo/bar/g }     # теперь это изменяет значения
```

Использование values с хешем, связанным с большим файлом DBM, создаст большой список, а в результате вы получите прожорливый процесс. Более практично в таком случае будет применить функцию each, которая обходит все записи хеша, не создавая при этом гигантского списка.

vec



vec *EXPR*, *OFFSET*, *BITS*

Обеспечивает компактное хранение списков беззнаковых целых. Эти целые плотно упаковываются, насколько это возможно, в обычную строку Perl. Строка в *EXPR* рассматривается как строка битов, составленная из произвольного числа элементов, зависящего от длины строки.

OFFSET указывает индекс конкретного элемента, который нас интересует. Синтаксис чтения и записи элемента одинаков, поскольку vec записывает или возвращает значение элемента в зависимости от того, в каком контексте она используется – левостороннего или правостороннего значения.

BITS задает ширину каждого элемента в битах и должно быть степенью двойки: 1, 2, 4, 8, 16 или 32 (на некоторых платформах также 64). (Если применить значение, нарушающее это правило, возникает исключение.) Поэтому каждый элемент может содержать целое в диапазоне 0..(2*BITS*)-1. Для меньших размеров в каждый байт упаковывается столько элементов, сколько возможно. Когда *BITS* равно 1, в каждом байте хранится 8 элементов. Когда *BITS* равно 2 – четыре. Когда *BITS* равно 4 – два элемента (обычно их называют полубайтами или «нибблами» (nybbles)). И так далее. Целые, занимающие больше одного байта, хранятся в прямом (big-endian) порядке.

Список целых чисел без знака можно хранить в одной скалярной переменной, присваивая их по отдельности вызовом функции vec. (Если *EXPR* не является левосторонним значением, возникает исключение.) В следующем примере все элементы имеют ширину 4 бита:

```
$bitstring =
$offset = 0;

for my $num (0, 5, 5, 6, 2, 7, 12, 6) {
    vec($bitstring, $offset++, 4) = $num;
}
```

При попытке произвести запись за пределами конца строки, Perl сначала увеличит строку, дополнив ее необходимым количеством нулевых байтов.

Векторы, хранимые в скалярной переменной, можно последовательно извлекать, задавая правильное смещение *OFFSET*.

```
$num_elements = length($bitstring)*2; # 2 элемента в байте

for my $offset (0 .. $num_elements-1) {
    say vec($bitstring, $offset, 4);
}
```

Если выбранный элемент находится за пределами конца строки, функция возвращает значение 0.

Строки, создаваемые vec, можно также обрабатывать логическими операторами |, &, ^ и ~. Эти операторы выполняют операции над битовыми строками, если оба операнда являются строками. См. соответствующие примеры в главе 3, в разделе «Поразрядные операторы».

Если *BITS* == 1, может быть создана битовая строка для записи последовательностей битов в один скаляр. Порядок таков, что vec($bitstring,0,1) гарантированно попадает в младший бит первого байта строки.

```
@bits = (0,0,1,0, 1,0,1,0, 1,1,0,0, 0,0,1,0);

$bitstring = "";
$offset = 0;

for my $bit (@bits) {
    vec($bitstring, $offset++, 1) = $bit;
}

say "$bitstring";       # "TC", т.е. '0x54', '0x43
```

Битовая строка может транслироваться в строку из единиц и нулей (и также в обратную сторону) путем передачи шаблона "b*" в pack или unpack. Можно также использовать pack с шаблоном "b*" для создания битовой строки из строки, состоящей из единиц и нулей. Порядок совместим с тем, который предполагает vec.

```
$bitstring = pack "b*", join(q(), @bits);
say "$bitstring";       # "TC", то же, что раньше
```

При помощи unpack можно организовать извлечение списка нулей и единиц из битовой строки.

```
@bits = split(//, unpack("b*", $bitstring));
say "@bits";            # 0 0 1 0 1 0 1 0 1 1 0 0 0 0 1 0
```

Если известна точная длина строки в битах, ее можно указать вместо "*".

Дополнительные примеры использования битовых строк, создаваемых посредством vec, можно найти в описании select. Примеры работы с двоичными данными более высокого уровня можно найти в описаниях pack и unpack.

wait

$! $? X U

```
wait
```

Ждет завершения любого порожденного процесса и возвращает его PID или -1, если порожденных процессов не осталось (такое возможно в некоторых системах, где порожденные процессы удаляются автоматически). Код завершения возвращается в $?, как сказано в описании system. Чтобы избежать превращения порожденных процессов в зомби, следует вызывать эту функцию или waitpid.

Если порожденный процесс стартовал, а функция wait его не обнаружила, возможно, вы вызывали system, закрытие канала или обратные апострофы между fork и wait. Эти конструкции тоже выполняют *wait*(2) и могли закрыть порожденный процесс. Во избежание этого применяйте waitpid.

waitpid

$! $? X U

```
waitpid PID, FLAGS
```

Ждет завершения конкретного порожденного процесса и возвращает его PID после завершения; -1, если порожденных процессов не осталось, или 0, если флаги

FLAGS задают неблокирующий вызов, а процесс еще не завершен. Код завершения процесса возвращается в $?, как сказано в описании system. Чтобы получить допустимые значения флагов, необходимо импортировать группу тегов импорта ":sys_wait_h" из модуля POSIX. Вот пример неблокирующего ожидания всех незавершенных процессов-зомби.

```
use POSIX ":sys_wait_h";
do {
    $kid = waitpid(-1,&WNOHANG);
} until $kid == -1;
```

В системах, где не реализован ни один из системных вызовов *waitpid*(2) и *wait4*(2), в аргументе *FLAGS* можно передать только 0. Иными словами, функция позволяет дождаться завершения порожденного процесса с конкретным PID, но не позволяет сделать это в неблокирующем режиме.

В некоторых системах возвращаемое значение -1 может означать, что порожденные процессы удаляются автоматически, так как пользователь установил $SIG{CHLD} = "IGNORE".

wantarray

```
wantarray
```

Возвращает истинное значение, если контекст выполняемой в данное время подпрограммы ожидает списочного значения, и ложное в противном случае. Функция возвращает определенное ложное значение (""), если вызывающий контекст ожидает скаляра, и неопределенное ложное значение (undef), если вызывающий контекст ничего не ожидает; т.е. если функция находится в пустом контексте.

Вот примеры типичного использования:

```
return unless defined wantarray;        # больше не надо трудиться
my @a = complex_calculation();
return wantarray ? @a : \@a;
```

См. также caller. Эта функция в действительности должна была называться wantlist, но мы придумали это название, когда списочный контекст еще назывался контекстом массива.

warn

$!

```
warn LIST
warn
```

Генерирует сообщение об ошибке, выводя *LIST* в STDERR подобно функции die, но не пытается завершить работу или возбудить исключение. Например:

```
warn "Отладка включена" if $debug;
```

Если список *LIST* пуст, а $@ уже содержит значение (обычно от предыдущего вызова eval), вслед за $@ в STDERR выводится строка "\t...caught". (Напоминает способ распространения ошибки функцией die, за исключением того, что warn не распространяет – не возбуждает повторно – исключение.) Если передана пустая строка сообщения, выводится текст "Warning: Something's wrong" (Внимание: произошла какая-то ошибка).

Как и в die, если передаваемые строки не заканчиваются символом перевода строки, автоматически добавляются сведения о файле и номере строки. Функция warn не связана с ключом командной строки Perl *-w*, но может применяться совместно с ней, например, когда требуется эмулировать встроенную функцию:

```
warn "Надвигается беда\n" if $^W;
```

Сообщение не выводится, если установлен обработчик $SIG{__WARN__}. Обработчик может поступить с сообщением так, как посчитает нужным. Например, простое предупреждение можно повысить до звания исключения:

```
local $SIG{__WARN__} = sub {
    my $msg = shift;
    die $msg if $msg =~ /isn't numeric/;
};
```

Поэтому в общем случае обработчик должен вывести предупреждение, столкнувшись с ситуацией, к которой оказался не готов, путем повторного вызова warn. Это совершенно безопасно и не породит бесконечный цикл, потому что обработчики __WARN__ не вызываются из обработчиков __WARN__. Такой режим несколько отличается от действия обработчиков $SIG{__DIE__} (которые не подавляют текст ошибки, но могут повторно вызвать die, чтобы изменить его).

Обработчик __WARN__ предоставляет мощный способ подавить все предупреждения, даже так называемые обязательные. Иногда обработчик требуется «завернуть» в блок BEGIN{} для выполнения на этапе компиляции:

```
# подавить *все* предупреждения этапа компиляции
BEGIN { $SIG{__WARN__} = sub { warn $_[0] if $DOWARN } }
my $foo = 10;
my $foo = 20;       # нет предупреждения о повторном объявлении my $foo,
                    # но вы сами этого хотели!

# до сих пор никаких предупреждений этапа компиляции или выполнения
$DOWARN = 1         # *не* встроенная переменная

# с этого места предупреждения этапа выполнения включены
warn "\$foo is alive and $foo!";   # выводится
```

Читайте в описании прагмы warnings о том, как управлять предупреждениями с лексической областью видимости. Другие способы вывода предупреждений реализованы функциями carp и cluck модуля Carp.

write




```
write FILEHANDLE
write
```

Выводит форматированную запись (возможно, из нескольких строк) в заданный дескриптор файла, используя формат, связанный с этим дескриптором, – см. раздел «Переменные формата» главы 26. По умолчанию формат, связанный с дескриптором файла, носит такое же имя. Однако формат для дескриптора файла можно изменить, модифицировав переменную $~ после выполнения select для этого дескриптора:

```
$old_fh = select(HANDLE);
$- = "NEWNAME";
select($old_fh);
```

или сказав:

```
use IO::Handle;
HANDLE->format_name("NEWNAME")
```

Поскольку форматы помещаются в пространство имен пакета, может потребоваться полностью квалифицировать имя формата, если объявление format находится в другом пакете:

```
$- = "OtherPack::NEWNAME";
```

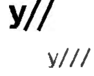
Обработка верхнего колонтитула страницы производится автоматически: если на текущей странице недостаточно места для форматированной записи, страница перемещается выводом символа разрыва страницы, для заголовка новой страницы используется специальный формат верхнего колонтитула страницы, а затем выводится запись. Число строк, подлежащих выводу на текущей странице, хранится в переменной $-, которую можно установить в 0, чтобы форсировать вывод новой страницы при следующем вызове write. (Может потребоваться сначала выполнить select для дескриптора файла.) По умолчанию именем формата верхнего колонтитула является имя дескриптора файла с добавлением "_TOP", но формат для дескриптора файла можно изменить установкой переменной $^ после выполнения select для этого дескриптора файла, либо так:

```
use IO::Handle;
HANDLE->format_top_name("NEWNAME_TOP");
```

Если дескриптор *FILEHANDLE* не задан, вывод производится в текущий дескриптор выходного файла, каковым изначально является STDOUT, но может быть изменен оператором select с одним аргументом. Если *FILEHANDLE* является выражением, оно будет вычислено на этапе выполнения, и его результат послужит фактическим аргументом *FILEHANDLE*.

Если указанный формат или формат верхнего колонтитула не существует, возникает исключение.

Функция write не является антиподом функции read. К несчастью. Для простого вывода строк пользуйтесь функцией print. Если вы открыли это описание в поисках способа обойти стандартную систему ввода/вывода, обратитесь к функции syswrite.



y//

```
y///
```

Оператор транслитерации (исторически называемый также трансляцией), известный еще как tr///. См. главу 5.

28

Стандартная библиотека Perl

Стандартный дистрибутив Perl включает значительно больше, чем просто исполняемый файл *perl*, который запускает ваши сценарии. В дистрибутив также входят сотни модулей с кодом, который можно использовать повторно, которые мы называем *Стандартной Библиотекой Perl.* Стандартные модули доступны всюду, поэтому программу, обращающуюся к ним, можно запускать везде, где установлен Perl, без дополнительных шагов при установке.

Однако следует заметить, что не везде, где имеется *perl*, присутствует стандартная библиотека Perl. Perl поддерживает множество разных платформ, и вы вполне можете столкнуться с такими операционными системами, производители которых изменяют состав библиотеки Perl. Некоторые производители расширяют ее, добавляя дополнительные модули и инструменты. Некоторые могут дополнять отдельные модули, улучшая совместимость со своими платформами (и, хочется верить, публикуют свои правки в общем репозитории). Однако есть и такие, кто удаляет те или иные модули. Если вы обнаружите, что в стандартной библиотеке отсутствует какая-то ее часть, сообщите об этом поставщику или установите Perl самостоятельно.

В предыдущих изданиях этой книги мы перечисляли все модули в стандартной библиотеке, сопровождая их небольшими описаниями. В этом издании мы убрали эти описания и вместо этого показываем, как их найти самостоятельно. Однако в главе 29 мы сохранили описания всех прагм.

Библиотечное дело

Приведем краткий обзор используемой терминологии. Мы, как и все остальное сообщество, достаточно свободно применяем эти термины, потому что понятия часто перекрываются или описывают смежные области, но иногда точность важна.

namespace

Пространство имен (*namespace*) представляет собой место, предназначенное для того, чтобы не путать имена, хранимые в нем, с именами в других пространствах имен. В результате масштабы проблемы уменьшаются – нам остается не

путать собственно пространства имен. Есть два способа различать пространства имен: давать им уникальные имена или уникальные местоположения. Perl позволяет сделать и то и другое: именованные пространства имен называются *пакетами*, а неименованные пространства имен называются *областями лексической видимости*. Поскольку области лексической видимости не могут быть больше, чем файл, и поскольку стандартные модули имеют размер файла (как минимум), отсюда следует, что все интерфейсы модулей должны использовать именованные пространства имен (пакеты), если с ними будет работать кто-то извне файла.

package

Пакет (*package*) – это стандартный механизм Perl для объявления именованного пространства имен. Он обеспечивает простой способ группировать связанные функций и переменные. Точно так же, как два каталога могут содержать (различные) файлы с именем *Amelia*, две различные части программы Perl могут содержать каждая свою переменную `$Amelia` или функцию `&Amelia`. Даже если этим переменным или функциям даны, казалось бы, одинаковые имена, эти имена находятся в различных пространствах имен, определяемых объявлениями `package`. Имена пакетов служат для идентификации модулей и классов, как описано в главах 11 и 12.

library

Термин *библиотека* (*library*), к несчастью, довольно перегружен в культуре Perl. В настоящее время этим термином мы обычно обозначаем все множество модулей Perl, установленных на данной машине.

Традиционно библиотекой в Perl назывался еще и файл, содержащий набор подпрограмм, служащих некоей общей цели. Такой файл часто имеет расширение *.pl*,[1] сокращение от «perl library». Мы по-прежнему используем это расширение для некоторых фрагментов кода Perl, загружаемых с помощью `do FILE` или `require`. Не будучи полноценным модулем, библиотечный файл обычно объявляет себя как отдельный пакет, чтобы связанные переменные и подпрограммы могли храниться вместе и случайно не помешали другим переменным вашей программы. Обязательного расширения не существует; помимо *.pl* иногда встречаются другие, как разъясняется далее в этой главе. В большинстве ситуаций на смену этим простым неструктурированным файлам пришли модули.

module

Модуль Perl – это библиотечный файл, удовлетворяющий неким особым соглашениям, которые позволяют одному или нескольким файлам, реализующим этот модуль, подключаться к программе посредством единственного объявления `use` на этапе компиляции. Имена файлов модулей всегда должны завершаться расширением *.pm*, поскольку это предполагается объявлением `use`. Объявление `use` также транслирует разделитель пакетов `::` в тот символ, который выполняет в данной системе функции разделителя каталогов, чтобы структура каталогов библиотеки Perl пользователя соответствовала структуре его пакета. Глава 11 описывает, как создавать собственные модули Perl.

---

[1] Да, некоторые используют это расширение для программ. Мы не видим в этом ничего плохого, если это согласуется с вашим образом мышления или если операционная система принуждает к этому.

class

Класс – это просто модуль, в котором реализованы методы объектов, связанных с именем пакета модуля. Если вас интересуют объектно-ориентированные модули, откройте главу 12.

pragma

Прагма (*pragma*) – это специальный модуль, который приводит в действие скрытые механизмы Perl. См. главу 29, где описаны прагмы стандартной библиотеки.

extension

Расширение (*extension*) – это модуль Perl, который помимо загрузки файла *.pm* загружает также общую библиотеку, реализующую семантику модуля на C или C++.

program

Программа (*program*) Perl представляет собой код, предназначенный для выполнения в качестве независимой сущности; ее называют также *сценарием* (*script*), когда хотят сказать, что не следует ожидать от нее слишком многого, *приложением* (*application*) – если она большая и сложная, *выполняемым модулем* (*executable*), когда вызывающему ее безразлично, на каком языке она написана, или *решением в масштабе предприятия* (*enterprise solution*) – если она обошлась в целое состояние. Программы Perl могут существовать в виде исходного кода, байт-кода или машинного кода. Если это можно запустить из командной строки, мы называем это программой.

distribution

Дистрибутив (*distribution*) – это архив, содержащий *сценарии*, *библиотеки* или *модули* с комплектом испытательных тестов, документацией и сценариями, выполняющими установку. Когда говорят: «получить модуль из CPAN», в действительности подразумевается: «получить дистрибутив». См. главу 19.

Обзор библиотеки Perl

Можно сэкономить огромное количество времени, предприняв попытку ознакомиться со стандартной библиотекой Perl, потому что совершенно незачем изобретать колесо заново. Предупреждаем, однако, что эта коллекция невероятно обширна. Некоторые библиотеки могут быть крайне полезны, другие могут оказаться совершенно неподходящими для ваших нужд. Например, тому, кто пишет на стопроцентно чистом Perl, модули, поддерживающие динамическую загрузку расширений C и C++, не сильно помогут.

Perl ожидает найти библиотечные модули где-то на маршруте поиска подключаемых библиотек, `@INC`. Этот массив задает упорядоченный список каталогов, в которых Perl осуществляет поиск, когда загружается какой-нибудь библиотечный код с использованием ключевых слов `do`, `require` или `use`. Перечень этих каталогов легко получить, выполнив Perl с ключом *-V* (от verbose) или такой простой код:

```
% perl -le "print for @INC"
/usr/local/lib/perl5/site_perl/5.14.2/darwin-2level
/usr/local/lib/perl5/site_perl/5.14.2
/usr/local/lib/perl5/5.14.2/darwin-2level
```

```
/usr/local/lib/perl5/5.14.2
```

Это лишь пример возможного вывода. В каждой установке Perl прописаны свои пути. Важно отметить, однако, что хотя содержание может отличаться в зависимости от поставщика и политики установки, которой придерживается сервер, можно полагаться на то, что все стандартные библиотеки будут установлены вместе с Perl. Это один из вариантов, когда Perl был установлен вручную. При другом способе установки Perl в той же системе можно получить совершенно другой ответ:

```
% /usr/bin/perl -le "print for @INC"
/Library/Perl/5.12/darwin-thread-multi-2level
/Library/Perl/5.12
/Network/Library/Perl/5.12/darwin-thread-multi-2level
/Network/Library/Perl/5.12
/Library/Perl/Updates/5.12.3
/System/Library/Perl/5.12/darwin-thread-multi-2level
/System/Library/Perl/5.12
/System/Library/Perl/Extras/5.12/darwin-thread-multi-2level
/System/Library/Perl/Extras/5.12
```

Этот вывод, полученный в Mac OS X 10.7, существенно отличается в смысле местоположения модулей. Здесь стандартная библиотека хранится в каталоге */System*, а обновления помещаются ближе к началу списка @INC. При обновлении Mac OS X обновления не затрут стандартную библиотеку, а попадут в другой каталог, определяемый производителем системы. Устанавливаемые вами модули окажутся в каталоге */Library*, поэтому обновление операционной системы не затронет ваших изменений. Однако при установке можно указать другой каталог (см. главу 19).

Если заглянуть в эти каталоги, можно обнаружить те же самые модули, но другие их версии, а также дополнительные модули. Некоторые поставщики вносят собственные правки в стандартную библиотеку. При этом они могут изменять или не изменять номера версий. Если вам кажется, что поведение вашего Perl отличается от поведения на других платформах, проверьте, действительно ли на других платформах Perl действует иначе.

Узнать местоположение модуля можно посредством команды *perldoc -l*:

```
% perldoc -l MODULE
/usr/local/lib/perl5/site_perl/5.14.2/MODULE
```

Внутри программы переменная %INC хранит имена уже загруженных модулей и пути к ним. Ключами являются пространства имен, преобразованные в пути к файлам, такие как *Unicode/UCD.pm*, а значениями – пути к файлам модулей. Подробности см. в главе 25.

Здесь уместно будет упомянуть одну из проблем, связанных с загрузкой модулей. Perl использует первое соответствие, найденное в @INC. Он не пытается найти последнюю версию или наиболее подходящую версию, и на практике нет достаточно удобного способа заставить *perl* продолжить поиск, кроме как самостоятельно реализовать процесс поиска и добавить его в начало @INC[1] или создать новый путь к библиотеке, куда помещать ссылки на «лучшие» версии модулей в других ката-

[1] Пример такой реализации можно найти в модуле inc::latest.

логах. Это не очень удобное решение, требующее осторожности и внимания. Например, если кто-то установит PERL5LIB, следует ли выбирать версии, найденные по указанному пути, или продолжить поиск в других каталогах?

Перекличка модулей

Предыдущее издание включало полный перечень модулей, составляющих стандартную библиотеку, но эта книга и так уже достаточно раздута, чтобы включать в нее еще несколько десятков страниц с простым перечислением модулей, тем более, что с помощью *perlmodlib* вы легко сможете сделать то же самое. Если результат работы этой команды вам не нравится, вы можете составить список самостоятельно, отыскивая файлы с расширением *.pm* и извлекая непустую строку после =head1 NAME:

```
use v5.10;

use File::Find;

my %names;
my $wanted = sub {
    return unless /\.pm\z/;
    open(my $fh, "<", $File::Find::name)
        || die "can't open $File::Find::name: $!";
    OUTER: while( <$fh> ) {
        next unless /\A =head1 \s+ NAME/x;
        while( <$fh> ) {
            next if /\A \s* \z/x;
            / (?<name>\S+) \s* -+ \s* (?<desc>.*) /x;
            $names{ $+{name} } = $+{desc};
            last OUTER;
        }
    }
};

find($wanted, @INC);

for my $name (sort keys %names) {
    printf "%-25s - %s\n", $name, $names{$name};
}
```

В Perl версии v5.14 этот сценарий найдет около 500 пространств имен:

```
AnyDBM_File            - provide framework for multiple DBMs
App::Cpan              - easily interact with CPAN from
                       - the command line
App::Prove             - Implements the C<prove> command.
... множество других
warnings               - Perl pragma to control optional warnings
warnings::register     - warnings import function
writemain              - write the C code for perlmain.c
```

Этот же список можно создать иначе. Модуль Module::CoreList, входящий в стандартную библиотеку Perl, знает, что входит в состав Perl. Он включает утилиту *corelist*. Чтобы найти версии, известные ему, запустите утилиту с ключом *-v*:

```
% corelist -v
5
5.000
5.001
5.002
...
v5.14.0
v5.14.1
```

Если указать номер версии с ключом *-v*, будет выведен список всех модулей, поставляемых с этой версией Perl:

```
% corelist -v 5.14.1

The following modules were in perl 5.14.1 CORE

AnyDBM_File 1.00
App::Cpan 1.5701
App::Prove 3.23
...множество других...
version 0.88
vmsish 1.02
warnings 1.12
warnings::register 1.02
```

Эта утилита может также выводить хронологию развития модулей, если применить ее с ключом *-a*:

```
% corelist -a Archive::Extract
Archive::Extract was first released with perl v5.9.5
  v5.9.5 0.22_01
  v5.10.0 0.24
  v5.10.1 0.34
  ...
  v5.14.0 0.48
  v5.14.1 0.48
```

Если потребуется узнать первую версию Perl, содержавшую модуль, запустите утилиту без ключей:

```
% corelist Module::CoreList
Module::CoreList was first released with perl v5.8.9
```

Ключ *-d* сообщает первую версию Perl, куда был включен модуль. Например, модуль `Module::CoreList` был включен в состав стандартной библиотеки в Perl v5.9.2:

```
% corelist -d Module::CoreList
Module::CoreList was first released with perl v5.9.2
```

Ключ *-d* означает «date» (дата), так что не запутайтесь. Версия Perl v5.9.2 вышла раньше версии v5.8.9, поэтому результаты выглядят немного странно.

Будущее стандартной библиотеки

Существуют две точки зрения на будущее стандартной библиотеки Perl. Первая ратует за то, чтобы поместить в стандартную библиотеку как можно больше модулей, чтобы иметь возможность создавать приложения с применением любых

модулей и распространять их, не требуя от пользователей устанавливать дополнительные модули. Другая нацелена на создание минимального дистрибутива Perl, содержащего лишь самые необходимые модули, с тем, чтобы устанавливать дополнительные модули из CPAN.

Каждая точка зрения дает определенные выгоды. Большая библиотека несет массу преимуществ для пользователей. Им не нужно обременять своих системных администраторов установкой дополнительных модулей, так как модули уже входят в состав Perl. Маленькая библиотека упрощает труд участников группы Perl 5 Porters, так как им меньше придется сталкиваться с проблемами переносимости модулей, и они смогут уделять больше внимания другим задачам.

Некоторые модули живут *двойной жизнью*, в том смысле, что они развиваются двумя путями. Одна версия находится непосредственно в репозитории Perl, а другая – в CPAN. Это позволяет исправлять проблемы в модулях быстрее, чем выходят новые версии Perl. Когда приходит время выхода новой версии Perl, сопровождающие лица просто переносят изменения из версии CPAN в исходные тексты Perl. Иногда версия в репозитории Perl исправляется раньше. В этом случае разработчики в CPAN переносят изменения в свою версию.

Долгие годы этот процесс был достаточно сложным, поскольку структура версий CPAN существенно отличалась от структуры версий стандартной библиотеки, что усложняло автоматизацию процесса объединения. Помимо исправлений, сопровождающим лицам требовалось объединять наборы тестов с остальной частью тестов Perl, переносить дополнительные файлы в нужное место и т.д. Нельзя сказать, что эта задача вызывала у людей большой энтузиазм. Теперь же принято помещать дистрибутивы из CPAN в отдельный каталог в репозитории *perl* и тем самым упрощать внесение изменений в модуль. Такой подход может быть принят всеми уже к моменту, когда вы будете читать эту книгу. Поддержка модулей, живущих двойной жизнью, существенно улучшилась в последние несколько лет. Это упрощает включение дополнительных модулей в нестандартные дистрибутивы Perl.

Блуждая по дереву каталогов

Если просмотреть каталоги и подкаталоги в иерархиях `@INC`, можно обнаружить несколько различных типов установленных файлов. Имена большинства файлов оканчиваются суффиксом *.pm*, но встречаются еще суффиксы *.pl*, *.ph*, *.al* и *.so*. Наибольший интерес для вас должна представлять первая группа, поскольку расширение *.pm* указывает, что файл является подлинным модулем Perl. Более подробно о них чуть ниже.

Несколько файлов, имена которых оканчиваются суффиксом *.pl*, представляют собой старые библиотеки Perl, о которых мы говорили выше. Они включены для совместимости со старыми версиями Perl из 80-х и начала 90-х годов. Благодаря им код Perl, работавший, скажем, в 1990 году, должен по-прежнему нормально работать, не привлекая к себе внимания, даже если у вас установлена современная версия Perl. Создавая новый код, который использует стандартную библиотеку Perl, всегда следует по возможности предпочесть версию *.pm* любой версии *.pl*. Дело в том, что модули не так засоряют пространство имен, как это делают старые файлы *.pl*. Поскольку Perl продолжает развиваться, члены группы Perl 5 Porters удалили некоторые из этих файлов, возложив решение соответствующих задач на модули или вынуждая вас посетить CPAN, чтобы получить нужные библиотеки.

Одно замечание об использовании расширения *.pl*: оно означает библиотеку Perl, а не программу Perl. Хотя *.pl* иногда применяется для обозначения программ на Perl на веб-серверах, чтобы отличать исполняемые модули и статические файлы в пределах одного каталога, мы предлагаем вместо этого расширения употреблять для обозначения исполняемых программ Perl суффикс *.plx*. (Аналогичный совет относится к операционным системам, которые выбирают интерпретатор, исходя из расширения имени файла.) Или не использовать расширение совсем, так как *perl* не требует его. Он будет счастлив выполнить файл *hello.rb*, при условии, что файл содержит программный код на языке Perl.[1]

Файлы с расширениями *.al* – это небольшие фрагменты более крупных модулей, они автоматически загружаются при обращении к их родительскому *.pm*-файлу. Если модуль построен с применением стандартного инструмента *h2xs* (см. главу 19), поставляемого с Perl (и если не был указан флаг Perl *-A*), то процедура `make install` воспользуется модулем `AutoLoader`, чтобы создать эти файлы *.al* без вашего участия.

Файлы *.ph* были созданы стандартной программой *h2ph*, несколько устаревшим, но иногда все еще необходимым инструментом, который старается транслировать директивы препроцессора C в Perl. Полученные файлы *.ph* содержат константы, иногда необходимые функциям низкого уровня, таким как `ioctl`, `fcntl` или `syscall`. (В настоящее время большинство этих значений может быть получено из стандартных модулей, таких как `POSIX`, `Errno`, `Fcntl` или `Socket`, намного более удобным и переносимым способом.) О том, как установить эти необязательные, но иногда важные компоненты, читайте в *perlinstall*.

И последнее расширение имени файла, с которым вы можете столкнуться в каталогах стандартной библиотеки, это *.so* (или другое применяемое в системе для обозначения общих библиотек). Эти файлы *.so* являются зависимыми от платформы фрагментами модулей расширений. Первоначально написанные на C или C++, эти модули были скомпилированы в динамически перемещаемый объектный код. Конечному пользователю, однако, не требуется знать об их существовании, поскольку они скрыты за интерфейсом модуля. Когда код пользователя говорит `require Module` или `use Module`, Perl загружает *Module.pm* и выполняет его, что позволяет модулю загрузить другие необходимые части, такие как *Module.so* или автоматически загружаемые компоненты *.al*. На самом деле модуль может загрузить все что угодно, включив 582 других модуля. Он может загрузить весь CPAN, если ему того захочется, а может быть, еще и архивы *freshmeat.net* за последние пару лет.

Модуль в Perl – это не просто статический фрагмент кода. Это активный агент, который определяет, как реализовать интерфейс в ваших интересах. Он может выполнять все стандартные соглашения, а может не выполнять. Он может делать любые вещи, с тем чтобы исказить смысл оставшейся части вашей программы, вплоть до того, что оттранслирует ее в SPITBOL. Такого рода крючкотворство считается совершенно допустимым, если оно хорошо документировано. При использовании такого модуля Perl вы соглашаетесь на его контракт, а не на стандартный контракт Perl.

Поэтому старайтесь читать мелкий шрифт.

---

[1] Одна из задач интерпретатора Parrot состоит в том, чтобы загрузить и выполнить файл *hello.rb*, даже если он содержит программный код на языке Ruby.

29

Модули прагм

Прагма (*pragma*) – это особый тип модуля, влияющий на этап компиляции программы. Некоторые модули прагм (или просто *прагмы*) могут влиять и на этап выполнения программы. Считайте их советами компилятору. Поскольку прагмы должны быть видимы на этапе компиляции, то и работают они только при вызове посредством `use` или `no`, потому что к тому времени, когда выполняется `require` или `do`, компиляция давно закончена.

Имена прагм принято записывать тольке строчными буквами, поскольку имена модулей в нижнем регистре зарезервированы для самого дистрибутива Perl. При написании собственных модулей используйте в имени модуля хотя бы одну заглавную букву, чтобы избежать конфликта с именами прагм.

В отличие от обычных модулей, действие прагм в большинстве случаев ограничено оставшейся частью самого внутреннего охватывающего блока, из которого они вызваны. Иными словами, они имеют лексическую область видимости, подобно переменным `my`. Обычно область лексической видимости внешнего блока охватывает любой вложенный в него блок, но внутренний блок может отменить прагму с лексической областью видимости из внешнего блока с помощью команды `no`:

```
use strict;
use integer;
{
    no strict "refs"; # разрешить символические ссылки
    no integer;       # возобновить арифметику с плавающей точкой
    # ..
}
```

Для среды компиляции Perl прагмы имеют намного большее значение, чем все другие модули, поставляемые с Perl. Трудно хорошо использовать компилятор, не зная, как передавать ему советы, поэтому в этой главе мы предпримем дополнительные усилия, чтобы описать прагмы.

Следует также знать, что мы часто применяем прагмы для прототипирования особенностей языка, которые затем превращаются в «реальный» синтаксис Perl. Поэтому в некоторых программах вы найдете устаревшие прагмы вроде `use attrs`,

функциональность которых сейчас поддерживается непосредственно синтаксисом объявления подпрограммы. Аналогично use vars находится в процессе замены объявлениями our. Мы не очень спешим ломать старые способы работы, но полагаем, что новые способы красивее.

Наконец, в конце этой главы мы покажем, как создавать собственные прагмы, действующие подобно штатным прагмам Perl.

attributes

```
sub afunc : method;
my $closure = sub : method { ... };

use attributes;
@attrlist = attributes::get(\&afunc);
```

Прагма attributes имеет два назначения. Первое – поддержка внутреннего механизма объявления *списков атрибутов* (*attribute lists*), которые являются необязательными свойствами, связываемыми с объявлениями подпрограмм и (в будущем) с объявлениями переменных. (Поскольку это внутренний механизм, обычно эта прагма не используется напрямую.) Второе назначение – извлечение этих списков атрибутов на этапе выполнения посредством функции attributes::get. В этом качестве attributes представляет собой обычный модуль, а не прагму.

В настоящее время обрабатывается лишь несколько встроенных атрибутов. Использование атрибутов, специфических для пакетов, служит экспериментальным механизмом расширения, описанным в разделе «Package-specific Attribute Handling» страницы руководства *attributes*(3).

Установка атрибутов происходит на этапе компиляции; попытка установить неизвестный атрибут является ошибкой компиляции. (Ошибка может перехватываться eval, но все равно прекращает компиляцию в этом блоке eval.)

В настоящее время реализованы только три встроенных атрибута подпрограмм: locked, method и lvalue. Более подробно они описаны в главе 7. В данное время атрибуты переменных, подобные реализованным для подпрограмм, не поддерживаются, но мы подумываем о некоторых вариантах, таких как constant.

Прагма attributes предоставляет две подпрограммы общего пользования. Вы можете запросить их импорт.

get

Возвращает (возможно, пустой) список атрибутов при передаче одного входного параметра, являющегося ссылкой на подпрограмму или переменную. Эта функция возбуждает исключение через Carp::croak, если получает неверные аргументы.

reftype

Действует сходным со встроенной функцией ref образом, но всегда возвращает основной встроенный тип данных для объекта ссылки, независимо от того, что он может быть «освящен» в какой-то пакет.

Реализация обработки атрибутов продолжает изменяться, поэтому за более полной информацией лучше обращаться к электронной документации, сопровождающей конкретную версию Perl.

autodie

```
use autodie;
```

Эта прагма превращает неудачи, случившиеся в ходе выполнения функций Perl, в фатальные ошибки, но только в своей лексической области видимости. Она замещает стандартные функции Perl, которые в случае ошибки возвращают ложное значение, их версиями, возбуждающими исключения. Текст сообщения, возвращаемого вместе с исключением, сохраняется в $@, что позволяет узнать причину ошибки:

```
eval {
    use autodie;
    open my $fh, "<:encoding(UTF-8)", $filename;
    my @lines = <$fh>;
    close $fh;
}
for ($@) {
    when (undef)  { }
    when ("open") { say "Сбой open";              }
    when (":io")  { say "Другая ошибка ввода/вывода";      }
    when (":all") { say "Другая ошибка autodie" }
    default       { say "Ошибка, не приводящая к вызову autodie"
}
```

Прагма может также замещать группы родственных функций, например только функций, связанных с вводом/выводом:

```
use autodie qw(:io);
```

Если действие этой прагмы нежелательно в какой-то внутренней области видимости, ее можно отключить:

```
no autodie;
```

autouse

```
use autouse "Carp" => qw(carp croak);
carp "this carp was predeclared and autoused";
```

Эта прагма реализует механизм загрузки модулей по требованию на этапе выполнения, т.е. в момент, когда вызывается некоторая функция из этого модуля. Это делается путем создания функции-заглушки, которая при вызове заменяет себя реальным вызовом. По духу она напоминает стандартные модули AutoLoader **и** SelfLoader**. Если сказать кратко, это «хак» для оптимизации производительности, помогающий программе на Perl запускаться (в общем случае) быстрее за счет отсутствия компиляции модулей, которые в данном прогоне могут быть ни разу не вызваны.**

Действие autouse **зависит от того, загружен ли модуль. Например, если модуль** Module **уже загружен, объявление:**

```
use autouse "Module" => qw(func1 func2($;$) Module::func3);
```

эквивалентно простому импорту двух функций:

```
use Module qw(func1 func2);
```

При этом предполагается, что Module определяет func2 с прототипом ($;$), а func1 и func3 не имеют прототипов. (Вообще говоря, также предполагается, что Module использует стандартный метод import из Exporter; в противном случае возникает фатальная ошибка.) Во всяком случае Module::func3 полностью игнорируется, поскольку функция предположительно уже объявлена.

Напротив, если на момент интерпретации прагмы autouse модуль Module еще не загружен, прагма объявляет, что функции func1 и func2 находятся в текущем пакете. Она также объявляет функцию Module::func3 (что можно рассматривать как умеренно антиобщественную деятельность, поскольку антиобщественные последствия отсутствия модуля Module еще сильнее). При вызове эти функции обеспечивают загрузку модуля Module, а затем заменяют себя вызовами только что загруженных фактических функций.

Поскольку прагма autouse перемещает часть выполнения программы с этапа компиляции на этап выполнения, это может иметь неприятные последствия. Например, если загружаемый по требованию модуль имеет инициализирующий код, выполнение которого предполагается на раннем этапе, инициализация может произойти слишком поздно. Также могут возникать невыявленные ошибки в коде, если важные проверки будут произведены на этапе выполнения вместо этапа компиляции.

В частности, если прототип, указанный в строке autouse, неверен, это не обнаружится до вызова соответствующей функции (что может произойти через месяцы или годы, если функция вызывается редко). Для частичного смягчения этой проблемы можно во время разработки писать код так:

```
use Chase;
use autouse Chase => qw(hue($) cry(&$));
cry "this cry was predeclared and autoused";
```

Первая строка обеспечивает раннее обнаружение ошибок при задании аргументов. Когда программа перейдет из стадии разработки в стадию эксплуатации, можно закомментировать обычную загрузку модуля Chase и оставить только вызов автозагрузки. В результате будут достигнуты надежность при разработке и производительность при эксплуатации.

base

```
use base qw(Mother Father);
```

Обеспечивает удобный способ определения производного класса на основе перечисленных родительских классов. Эта прагма почти вышла из употребления, так как большинство предпочитает пользоваться прагмой parent.

Приведенное объявление примерно эквивалентно следующему коду:

```
BEGIN {
    require Mother;
    require Father;
    push @ISA, qw(Mother Father);
}
```

Прагма base выполняет все необходимые вызовы require. Если в области видимости находится прагма strict "vars", вызов use base позволяет (в конечном итоге)

осуществить присваивание @ISA без необходимости вначале объявлять our @ISA. (Поскольку прагма base обрабатывается на этапе компиляции, лучше не пытаться самостоятельно изменять @ISA на этапе выполнения.)

Но помимо этого base имеет еще одно свойство. Если какой-либо именованный базовый класс использует механизм полей, описываемых в объявлении прагмы fields (упоминается далее в этой главе), тогда прагма base инициализирует специальные атрибуты полей пакета из базового класса. (Множественное наследование классов полей *не* поддерживается. Прагма base возбуждает исключение, если поля присутствуют более чем в одном именованном базовом классе.)

Любой еще не загруженный базовый класс будет автоматически загружен вызовом require. Однако необходимость вызова require для пакета базового класса определяется не обычным изучением %INC, а отсутствием глобальной переменной $VERSION в базовом пакете. Этот прием не позволяет Perl повторно пытаться (с неудачным исходом) загружать базовый класс, отсутствующий в собственном загружаемом файле (например, из-за того что он загружен как часть файла некоторого другого модуля). Если $VERSION не обнаружена после успешной загрузки файла, base определит $VERSION в базовом пакете, присвоив ей значение "-1, defined by base.pm". В последующих версиях прагмы эта строка может измениться.

bigint

```
use bigint;
```

Снимает некоторые архитектурные ограничения и позволяет работать с очень большими целыми числами, а также обрабатывать специальное значение NaN (Not a Number – не число):

```
use bigint;
say 2 ** 512;
```

Действие прагмы основано на перегрузке арифметических операторов с помощью модуля Math::BigInt. Эти операторы могут действовать значительно медленнее встроенных. Однако лучше подождать и получить правильный ответ, чем поспешить и получить неправильный.

Имеется возможность загружать разные библиотеки реализаций, которые могут отличаться производительностью. По умолчанию bigint использует реализацию на языке Perl, но вы можете загрузить более быструю библиотеку, если она у вас имеется:

```
use bigint lib => 'GMP';
```

Можно ограничить число значимых цифр в результате:

```
use bigint a => 2;
```

Или точность, определив порядок округления результата. Точность меньше 0 игнорируется:

```
use bigint p => -2; # до сотых долей (игнорируется)
use bigint p => 1;  # с округлением до 10
```

bignum

```
use bignum;
```

Снимает некоторые архитектурные ограничения и позволяет работать с очень большими числами (или с числами, имеющими большое количество знаков после запятой), а также обрабатывать специальное значение `NaN` (Not a Number – не число):

```
use bignum;
say sqrt(2);
```

Действие прагмы основано на перегрузке арифметических операторов с помощью модулей `Math::BigInt` и `Math::BigFloat`. См. `bigint`.

bigrat

```
use bigrat;
```

Снимает некоторые архитектурные ограничения и позволяет работать с рациональными числами (т.е. с дробями), сохраняя их в виде рациональных чисел, чтобы устранить потери точности (по крайней мере, до момента, когда потеря точности станет уже не критичной):

```
use bigrat;
say 1/2 + 1/3;      # 5/6
```

Действие прагмы основано на перегрузке арифметических операторов с помощью модулей `Math::BigInt` и `Math::BigRat`. См. `bigint`.

blib

Из командной строки:

```
% perl -Mblib program [args...]
% perl -Mblib=DIR program [args...]
```

Из программы Perl:

```
use blib;
use blib DIR;
```

Предназначена преимущественно для тестирования произвольных программ Perl с отсутствующими в системе версиями пакетов с помощью ключа командной строки Perl *-M*. При этом предполагается, что структура каталогов была создана с помощью стандартного модуля `ExtUtils::MakeMaker` или `Module::Build`.

Ищет структуру каталогов *blib*, начиная от каталога *DIR* (или от текущего, если аргумент *DIR* опущен), просматривая до пяти уровней вложенных подкаталогов. В случае неудачи предпринимает попытку поиска в родительском каталоге «..».

bytes

```
use bytes;
no bytes;
```

Отключает символьную семантику для оставшейся части лексической области видимости. Для отмены действия `use bytes` в текущей лексической области видимости может применяться прагма `no bytes`.

Perl обычно предполагает символьную семантику в присутствии символьных данных (данных из источника, для которого определена определенная кодировка символов).

Чтобы понять различие между символьной и байтовой семантикой, обратитесь к главе 6. Небесполезной может оказаться также поездка в Токио.

Эта прагма используется все реже и, скорее всего, она исчезнет в последующих версиях Perl. В Perl версии v5.14 документация прагмы `bytes` настоятельно не рекомендует использовать эту прагму. Если у вас есть строка, вы можете обрабатывать ее как последовательность символов, не беспокоясь о кодировке.

charnames

```
use charnames HOW;
print "\N{CHARNAME} is a funny character";

use charnames (); # функции времени выполнения,
                  # а не конструкции \N{} времени компиляции
```

Все формы, отличные от `use charnames ()`, включают интерполяцию именованных символов в строках с помощью обозначений \N{*CHARNAME*}:

```
use charnames ":full";
print "\N{GREEK SMALL LETTER SIGMA} называется сигмой.\n"

use charnames ":short";
print "\N{greek:Sigma} это прописная сигма.\n";

use charnames qw(cyrillic greek);
print "\N{sigma} это греческая сигма, а \N{be} это кириллическое б.\n"

use charnames ":full", ":alias" => {
    "WRY CAT"      => "CAT FACE WITH WRY SMILE",
    "AMELIA"       => "DROMEDARY CAMEL",
    "s with comma" => 0x0219,
};

# ":loose" поддерживается только в v5.16 и выше
use charnames ":loose";
```

Аргумент `:full` указывает, что *CHARNAME* следует считать полным именем, а поиск выполнять в списке стандартных имен символов Юникода. Аргумент `:short` в сочетании с *CHARNAME* вида *SCRIPT:CNAME*, указывает, что *CNAME* — это буква, а поиск следует выполнять в алфавите *SCRIPT*. Если присутствует аргумент `:loose` (и сценарий выполняется под управлением Perl v5.16 или более поздней версии), он действует точно как `:full`, за исключением того, что поиск имен выполняется без учета регистра символов, наличия пробельных символов и символов подчеркивания, по аналогии с именами свойств Юникода в регулярных выражениях.

Если *HOW* содержит конкретные названия алфавитов[1], *CHARNAME* считается буквой, и поиск выполняется в указанных алфавитах в указанном порядке. Чтобы найти *CHARNAME* в заданном алфавите *SCRIPTNAME*, производится поиск по шаблонам:

```
SCRIPTNAME CAPITAL LETTER CHARNAME
SCRIPTNAME SMALL LETTER CHARNAME
SCRIPTNAME LETTER CHARNAME
```

Если *CHARNAME* целиком находится в нижнем регистре, то вариант CAPITAL игнорируется. В противном случае игнорируется вариант SMALL.

```
use charnames "Greek";
print "\N{Sigma} \N{sigma} \N{final sigma}\n";     # Σ σ ς

use charnames "Latin";
print "\N{DZ} \N{D with small letter z} \N{dz}\n"; # Ǳ ǲ ǳ
```

Конструкция \N{*NAME*} интерпретируется только на этапе компиляции как особая форма строковой константы в двойных кавычках. Имя символа *NAME* должно быть литералом; внутри конструкции \N{*NAME*} нельзя использовать переменную. Аналогичная функциональность на этапе выполнения предоставляется функцией charnames::string_vianame, описываемой ниже.

Форма записи \N{U+*HEXDIGITS*}, где *HEXDIGITS* – шестнадцатеричное число, также вставляет символ Юникода в строку, но в отличие от всех остальных форм записи, \N{...} не требует прагмы charnames. Она вставляет символ с кодом (порядковым значением), равным указанному шестнадцатеричному числу. Например, \N{U+263A} соответствует символу Юникода, изображающему улыбающееся лицо (черное лицо на белом фоне). Эта форма записи не требует прагмы charnames, тогда как для форм записи с именами символов вроде \N{WHITE SMILING FACE} использование данной прагмы является обязательным условием.

Любая строка, содержащая \N{*CHARNAME*} или \N{U+*HEXDIGITS*}, автоматически получает семантику Юникода, даже если не используются функции для работы с Юникодом.

Для управляющих символов C0 и C1 (U+0000..U+001F, U+0080..U+009F) не существует официальных имен в Юникоде, но вы можете использовать имена, определяемые стандартом ISO 6429: LINE FEED, ESCAPE и т. д. или их аббревиатуры: LF, ESC и другие. Список часто используемых синонимов можно найти на странице *charnames* справочного руководства.

Если имя символа в конструкции \N{*NAME*} не обнаруживается, выводится предупреждение и выполняется замещение символом Юникода REPLACEMENT CHARACTER (U+FFFD).

Если в области действия прагмы bytes символ \N{*NAME*} не укладывается в один байт (т.е. если его порядковый номер превышает 255), возникает фатальная ошибка.

[1] Текущий список алфавитов в Юникоде можно найти в странице *perluniprops* справочного руководства.

Пользовательские имена символов

Существует возможность определять собственные имена символов, чтобы упростить их ввод или дать имена кодам, не имеющим их. Синонимы можно добавлять с помощью анонимных хешей:

```
use charnames ":alias" => {
    e_ACUTE => "LATIN SMALL LETTER E WITH ACUTE",
    "APPLE LOGO" => 0xF8FF, # код для частного использования
};
my $str = "\N{APPLE LOGO}";
```

или с помощью файла, хранящего список пар ключ/значение:

```
use charnames ":alias" => "pro"; # загляните в unicore/pro_alias.pl
```

Файл должен храниться в каталоге *unicore/* где-то в пути `@INC`**, а его имя должно оканчиваться строкой *_alias.pl*. Например, файл, упомянутый выше, будет иметь имя *unicore/pro_alias.pl*. Этот файл должен содержать простой плоский список Perl:**

```
(
    A_grave  => "LATIN CAPITAL LETTER A WITH GRAVE",
    A_circ   => "LATIN CAPITAL LETTER A WITH CIRCUMFLEX",
    A_diaer  => "LATIN CAPITAL LETTER A WITH DIAERESIS",
    A_dier   => "LATIN CAPITAL LETTER A WITH DIAERESIS",
    A_uml    => "LATIN CAPITAL LETTER A WITH DIAERESIS",
    A_tilde  => "LATIN CAPITAL LETTER A WITH TILDE",
    A_macron => "LATIN CAPITAL LETTER A WITH MACRON".
);
```

Оба метода автоматически вставляют `:full` **в качестве первого аргумента, если нет других аргументов. Аргумент** `:full` **можно также передать явно:**

```
use charnames ":full", ":alias" => "pro";
```

Имена можно давать даже символам, зарезервированным для частного использования. Например, после:

```
use charnames ":full", ":alias" => {
    "TENGWAR LETTER TINCO"  => 0xE000,
    "TENGWAR LETTER PARMA"  => 0xE001,
    "TENGWAR LETTER CALMA"  => 0xE002,
    "TENGWAR LETTER QUESSE" => 0xE003,
    "TENGWAR LETTER ANDO"   => 0xE004,
    ...
}
```

строковые константы и константы регулярных выражений, находящиеся в той же лексической области видимости, могут ссылаться на символы по этим именам:

```
if (/\N{TENGWAR LETTER TINCO}/) { ... }
```

Поиск на этапе выполнения

Данная прагма также предоставляет три функции для выполнения преобразований между именами символов и числовыми значениями во время выполнения,

а не во время компиляции, как конструкция \N{*CHARNAME*}. Эти функции перечислены ниже:

charnames::vianame

Принимает официальное имя, официальный синоним или пользовательский синоним и возвращает единственный код символа. Например, преобразует строку "LATIN SMALL LETTER A" в число 0x61.

charnames::string_vianame

Принимает строку, которая может быть официальным именем, официальным синонимом или именованной последовательностью, и возвращает строку. Например, строка "LATIN SMALL LETTER A" будет преобразована в "a". В случае именованных последовательностей возвращаемая строка (иногда) может содержать более одного символа.

charnames::viacode

Принимает целое число и возвращает соответствующий официальный синоним, и официальное имя, если синоним отсутствует. Например, преобразует число 0x61 в строку "LATIN SMALL LETTER A". Пользовательское имя возвращаются, только если отсутствует официальное имя, как, например, для кодов, зарезервированных для частного использования.

Эти функции не экспортируются, поэтому для работы с ними требуется указывать полные квалифицированные имена. Они также обеспечивают доступ к любым пользовательским синонимам на этапе выполнения программы. Следующий пример демонстрирует работу каждой из этих функций:

```
use v5.14;
use warnings;
use warnings FATAL => "utf8";
use open qw(:std :utf8);

use charnames ":full", ":alias" => {
    ecute        => "LATIN SMALL LETTER E WITH ACUTE",
    "APPLE LOGO" => 0xF8FF, # символ для частного использования
};

printf "U+%04X is named '%s'.\n", 0xE9  => charnames::viacode(0xE9);
printf "%s is code U+%04X.\n",    ecute => charnames::vianame("ecute");
printf "%s is string '%s'.\n",    ecute => charnames::string_vianame("ecute");

printf "U+%04X is named '%s'.\n",    0xF8FF => charnames::viacode(0xF8FF);
printf "%s is code U+%04X.\n", "APPLE LOGO" => charnames::vianame("APPLE LOGO");
printf "%s is string '%s'.\n", "APPLE LOGO" => charnames::string_vianame("APPLE LOGO");
```

Ниже представлен вывод этого сценария:

```
U+00E9 is named 'LATIN SMALL LETTER E WITH ACUTE'.
ecute is code U+00E9
ecute is string 'é'

U+F8FF is named 'APPLE LOGO'.
APPLE LOGO is code U+F8FF.
APPLE LOGO is string ''
```

Вы можете даже написать собственный модуль, работающий, как прагма charnames, но по-иному определяющий имена символов. Однако предназначенный для этого интерфейс все еще является экспериментальным, поэтому для получения последних сведений обратитесь к странице руководства.

constant

```
use constant BUFFER_SIZE => 4096;
use constant ONE_YEAR    => 365.2425 * 24 * 60 * 60;
use constant PI          => 4 * atan2 1, 1;
use constant DEBUGGING   => 0;
use constant ORACLE      => 'oracle@cs.indiana.edu';
use constant USERNAME    => scalar getpwuid($<);
use constant USERINFO    => getpwuid($<);

use constant {
    BUFFER_SIZE => 4096,
    ONE_YEAR    => 365.2425 * 24 * 60 * 60,
    PI          => 4 * atan2( 1, 1 ),
    DEBUGGING   => 0,
    ORACLE      => 'oracle@cs.indiana.edu',
    USERNAME    => scalar getpwuid($<),
    USERINFO    => getpwuid($<),
};

sub deg2rad { PI * $_[0] / 180 }

print "This line does nothing"     unless DEBUGGING;

# ссылки могут быть объявлены константами
use constant CHASH          => { foo => 42 };
use constant CARRAY         => [ 1,2,3,4 ];
use constant CCODE          => sub { "bite $_[0]\n" }

print CHASH->{foo};
print CARRAY->[$i];
print CCODE->("me");
print CHASH->[10]; # ошибка этапа компиляции
```

Объявляет именованный символ неизменяемой константой[1] с заданным скалярным или списочным значением. Значения вычисляются в списочном контексте. С помощью scalar можно принудительно установить скалярный контекст, как показано выше. Передавая ссылку на хеш, можно объявить множество констант единственной инструкцией use.

Поскольку имена констант не предваряются символом $, их нельзя непосредственно интерполировать в строки, заключенные в двойные кавычки, но можно делать это косвенно:

```
print "The value of PI is @{[ PI ]}.\n";
```

1 Реализованной как подпрограмма без аргументов, возвращающая одно и то же значение.

Поскольку списочные константы возвращаются как списки, а не как массивы, индексирование следует осуществлять посредством дополнительных круглых скобок, как и для любых других списочных значений:

```
$homedir = USERINFO[7];   # НЕВЕРНО
$homedir = (USERINFO)[7]; # ok
```

Хотя имена констант рекомендуется составлять только из заглавных букв (плюс символы подчеркивания между словами), чтобы выделить их и предотвратить конфликты с другими ключевыми словами и названиями подпрограмм, это всего лишь соглашение. Имена констант должны начинаться буквой или символом подчеркивания, но буква не обязана быть заглавной.

Константы не ограничиваются лексической областью видимости, в которой действует прагма. Они представляют собой просто подпрограммы без аргументов в таблице символов пакета, в котором объявлены. На константу *CONST* из пакета Other можно сослаться, как на Other::*CONST*. Читайте дополнительно о встраивании таких подпрограмм на этапе компиляции в разделе «Подставляемые функции-константы» главы 7.

Как и для всех директив use, выполнение use constant происходит на этапе компиляции. Поэтому объявление constant внутри условного оператора, например if ($foo) { use constant ... }, будет в лучшем случае вводить в заблуждение. Поскольку директива выполняется на этапе компиляции, Perl может заменить выражение его фактическим значением.

Если опустить значение символа, он будет давать значение undef в скалярном контексте и пустой список () в списочном. Но лучше объявлять такие значения явным образом:

```
use constant CAMELIDS   => ();
use constant CAMEL_HOME => undef;
```

Ограничения констант

Списочные константы в настоящее время не внедряются в код (inline), как скалярные константы. Невозможно также создать подпрограмму или ключевое слово с таким же именем, как у константы. И это, вероятно, Правильный Путь.

Нельзя объявить одновременно больше одной константы:

```
use constant FOO => 4, BAR => 5;      # НЕВЕРНО
```

В результате будет объявлена константа FOO, возвращающая список (4, "BAR", 5). Вместо этого нужно записать:

```
use constant FOO => 4
use constant BAR => 5;
```

или даже:

```
use constant {
    FOO => 4,
    BAR => 5,
};
```

Неприятности могут возникнуть в случае применения констант в контексте, в котором голые имена автоматически заключаются в кавычки. (Это относится

к любому вызову подпрограммы, не только к константам.) Например, нельзя сказать `$hash{CONSTANT}`, потому что *CONSTANT* будет интерпретироваться как строка. Используйте `$hash{CONSTANT()}` или `$hash{+CONSTANT}`, чтобы воспрепятствовать действию механизма расстановки кавычек. Аналогично, поскольку оператор `=>` заключает в кавычки свой левый операнд, если он является голым именем, правильной будет запись `CONSTANT() => "value"`, а не `CONSTANT => "value"`.

deprecate

```
use deprecate;
```

Базовые модули, помеченные как кандидаты на удаление из стандартной библиотеки, используют эту прагму для вывода предупреждения, что вместо них должны использоваться версии из CPAN. Если модуль не входит в состав стандартной библиотеки, эта прагма ничего не делает.

diagnostics

```
use diagnostics;          # включение для этапа компиляции
use diagnostics -verbose;

enable diagnostics;       # включение для этапа выполнения
disable diagnostics;      # выключение для этапа выполнения
```

Расширяет обычную сжатую диагностику и подавляет вывод повторяющихся предупреждений. Дополняет краткие версии более подробными пояснениями, которые можно найти в *perldiag*. Как и другие прагмы, действует на этапе компиляции, а не только на этапе выполнения.

Если поместить `use diagnostics` в начало программы, автоматически включается режим *-w* командной строки путем установки `$^W` в значение 1. Последующая компиляция будет осуществляться с расширенной диагностикой. Сообщения будут по-прежнему выводиться в `STDERR`.

Нельзя применять `no diagnostics` для отключения сообщений на этапе компиляции из-за взаимодействия между задачами этапа выполнения и этапа компиляции, а также потому, что это вообще плохая идея. Однако управлять диагностикой на этапе выполнения можно с помощью методов `disable` и `enable`. (Не забудьте сначала выполнить `use`, иначе невозможно будет использовать эти методы.)

Флаг `-verbose` предписывает выводить вступление к странице руководства *perldiag* перед какой-либо диагностикой. Можно установить переменную `$diagnostics::PRETTY` (перед тем, как выполнять `use`), чтобы генерировались улучшенные escape-последовательности для программ постраничного вывода вроде *less*(1) или *more*(1):

```
BEGIN { $diagnostics::PRETTY = 1 }
use diagnostics;
```

Предупреждения Perl, которые обнаруживает эта прагма, выводятся только один раз. Это удобно, если вы находитесь в цикле, повторяющем одно и то же сообщение (например, неинициализированное значение). Предупреждения, создаваемые вручную, например вызовами `warn` или `carp`, не фильтруются этим механизмом обнаружения дубликатов.

Вот несколько примеров использования прагмы diagnostics. Приводимый файл, несомненно, выдаст несколько ошибок как при компиляции, так и во время выполнения:

```
use diagnostics;
print NOWHERE "nothing\n";
print STDERR "\n\tThis message should be unadorned.\n";
warn "\tThis is a user warning";
print "\nDIAGNOSTIC TESTER: Please enter a <CR> here: "
my $a, $b = scalar <STDIN>;
print "\n";
print $x/$y;
```

А вот вывод:

```
Parentheses missing around "my" list at diag.pl line 6 (#1)
    (W parenthesis) You said something like

        my $foo, $bar = @_;

    when you meant

        my ($foo, $bar) = @_;

    Remember that "my", "our", "local" and "state" bind
    tighter than comma.

Name "main::y" used only once: possible typo at diag.pl line 8 (#2)
    (W once) Typographical errors often show up as unique variable names.
    If you had a good reason for having a unique name, then just mention
    it again somehow to suppress the message. The our declaration is
    provided for this purpose.

    NOTE: This warning detects symbols that have been used only once so
    $c, @c, %c, *c, &c, sub c{}, c(), and c (the filehandle or format)
    are considered the same; if a program uses $c only once but also uses
    any of the others it will not trigger this warning.

Name "main::b" used only once: possible typo at diag.pl line 6 (#2)
Name "main::NOWHERE" used only once: possible typo at diag.pl line 2 (#2)
Name "main::x" used only once: possible typo at diag.pl line 8 (#2)

print() on unopened filehandle NOWHERE at diag.pl line 2 (#3)
    (W unopened) An I/O operation was attempted on a filehandle that was
    never initialized. You need to do an open(), a sysopen(), or a socket()
    call, or call a constructor from the FileHandle package.

  This message should be unadorned.
  This is a user warning at diag.pl line 4.

DIAGNOSTIC TESTER: Please enter a<CR> here:

Use of uninitialized value $y in division (/) at diag.pl line 8, <STDIN>
    line 1 (#4) (W uninitialized) An undefined value was used as if it
    were already defined. It was interpreted as a "" or a 0, but maybe
    it was a mistake. To suppress this warning assign a defined value to
    your variables.
```

```
    To help you figure out what was undefined, perl will try to tell you
    the name of the variable (if any) that was undefined. In some cases
    it cannot do this, so it also tells you what operation you used the
    undefined value in. Note, however, that perl optimizes your program
    and the operation displayed in the warning may not necessarily appear
    literally in your program. For example, "that $foo" is usually
    optimized into "that ". $foo, and the warning will refer to the
    concatenation (.) operator, even though there is no . in your program.

Use of uninitialized value $x in division (/) at diag.pl line 8,
    <STDIN> line 1 (#4)

Illegal division by zero at diag.pl line 8, <STDIN> line 1 (#5)
    (F) You tried to divide a number by 0. Either something was wrong in
    your logic, or you need to put a conditional in to guard against
    meaningless input.

Uncaught exception from user code:
    Illegal division by zero at diag.pl line 8, <STDIN> line 1.
at diag.pl line 8
```

Диагностические сообщения извлекаются из страницы *perldiag* справочного руководства. Если обнаруживается обработчик `$SIG{__WARN__}`**, он будет действовать, но только после того как функция** `diagnostics::splainthis` **(обработчик** `$SIG{__WARN__}` **в прагме) закончит свою работу с вашими предупреждениями. В настоящее время Perl не поддерживает стеки обработчиков, так что это лучшее, что мы сейчас можем сделать. Если вам уж очень любопытно, что же там такое перехватывается, установите переменную** `$diagnostics::DEBUG`**:**

```
BEGIN { $diagnostics::DEBUG = 1 }
use diagnostics;
```

encoding

```
use encoding ENCODING;
use encoding "euc-jp";
```

Эта прагма исходно предназначалась для того, чтобы дать возможность писать исходные тексты на языке Perl в любой кодировке *ENCODING* и позволить Perl безошибочно преобразовывать строки символов в программе, а также выводить сообщения потока стандартного вывода и потока ошибок в указанной кодировке. Однако она никогда не работала правильно и, вероятно, никогда уже не будет. Вместо этого мы рекомендуем вам преобразовывать исходные тексты программ в кодировку UTF-8, после чего включать объявление `use utf8` **в начале файла. Для установки кодировки стандартных потоков ввода/вывода используйте прагму** `open` **или функцию** `binmode`**.**

feature

```
use feature ":5.10"; # это целый "пакет особенностей"
use feature qw(say state switch unicode_strings);

{
```

```
    no feature qw(say);
    ...;
}
```

Perl готовится к наступлению будущего, вводя новые ключевые слова и возможности и предоставляя доступ к ним посредством прагмы feature. Эта прагма включает или выключает возможности языка в лексической области видимости. Включить или выключить те или иные возможности можно с помощью тега версии или имени возможности.

say

Включает ключевое слово say, действующее подобно print, но автоматически добавляющее символ перевода строки.

state

Включает ключевое слово state, позволяющее использовать в подпрограммах переменные, сохраняющие свои значения между вызовами.

switch

Включает «заряженную» инструкцию given-when.

unicode_strings

Эта возможность не является ключевым словом. Она принудительно придает всем строкам в лексической области видимости семантику Юникода. Она также придает эту семантику всем регулярным выражениям, компилируемым в области ее видимости, даже если эти регулярные выражения будут использоваться за ее пределами. См. раздел «The Unicode Bug» в странице *perlunicode* справочного руководства. Эта единственная особенность, которая не входит в пакет :v5.10, хотя она входит в пакеты :v5.12 и более поздних версий.

fields

Применение этой прагмы не рекомендуется начиная с Perl версии v5.10, и мы не станем поощрять ее использование, рассказывая о ней. Она появилась как способ объявлять поля класса, тип которых проверяется на этапе компиляции. Механизм работы fields опирался на возможность языка (ныне не существующую), известную под названием псевдохеш. Если вам доведется столкнуться с прагмой fields в унаследованном коде, вы всегда сможете прочитать о ней в документации, которая все еще содержит ее описание (во всяком случае, это так для Perl версии v5.14).

filetest

```
$can_perhaps_read = -r "file";     # использовать биты режима
{
    use filetest "access";         # стараться угадать
    $can_really_read = -r "file";
}
$can_perhaps_read = -r "file";     # использовать биты режима
```

Эта прагма с лексической областью видимости указывает компилятору на необходимость изменить режим работы унарных операторов тестирования файлов -r,

-w, -x, -R, -W и -X, описанных в главе 3. В режиме по умолчанию эти операторы используют разряды режима, возвращаемые семейством вызовов stat. Однако это не всегда правильно, например, если файловая система поддерживает ACL (Access Control Lists – списки управления доступом). В таких средах, как AFS, где это имеет значение, прагма filetest помогает операторам разрешений возвращать результаты, лучше согласующиеся с другими инструментами.

Применение этой прагмы может несколько снижать производительность операторов проверки файлов, потому что в некоторых системах расширенную функциональность приходится эмулировать.

Предупреждение: всякие мысли о применении проверок файлов для защиты данных совершенно бессмысленны. Здесь открываются широкие возможности для возникновения «ситуации гонки», так как невозможно гарантировать, что права доступа не изменятся в промежутке между тестированием и реальным действием. Заботясь о безопасности, не следует использовать операторы тестирования файлов, чтобы решить, *будет ли* нечто работать. Нужно просто выполнить фактическую операцию, а затем проверить, успешно ли она завершилась (это нужно делать в любом случае). См. раздел «Обработка ошибок синхронизации» главы 20.

if

```
use if CONDITION, MODULE => IMPORTS;

use if $^O =~ /MSWin/, "Win32::File";

use if $^V >= 5.010, parent => qw(Mojolicious::UserAgent),
use if $^V <  5.010, base   => qw(LWP::UserAgent);
```

Прагма if позволяет организовать условную загрузку модуля. Она не позволяет реализовать загрузку модуля с минимально допустимым номером версии. После имени модуля можно указать список импортируемых возможностей.

inc::latest

```
use inc::latest "Module::Build";
```

Некоторые авторы модулей распространяли зависимости, включая их в каталог *inc* в дистрибутиве. Если требовалось использовать определенную версию модуля Module::Build, например, они устанавливали ее в каталог *inc* в своем дистрибутиве и отдавали предпочтение этой версии перед любой другой. До того, как инструментальные средства Perl научились понимать configure_requires, эта прагма позволяла запускать процедуру сборки с модулями внутри дистрибутива.

Прагма inc::latest сообщает Perl о необходимости загрузить версию из каталога *inc*, но только если эта версия больше установленной в @INC.

integer

```
use integer;
$x = 10/3;
# $x теперь 3, а не 3.33333333333333333
```

Эта прагма с лексической областью видимости указывает компилятору, что с этого места и до конца охватывающего блока он должен использовать операции с целыми числами. Во многих системах это не играет особой роли для большинства вычислений, но на тех редких архитектурах, где нет аппаратной поддержки действий над числами с плавающей запятой, производительность может резко измениться.

Обратите внимание, что прагма оказывает воздействие на некоторые числовые операции, но не на сами числа. Например, если выполнить такой код:

```
use integer;
$x = 1.8;
$y = $x + 1
$z = -1.8;
```

в результате получится: `$x == 1.8`, `$y == 2` и `$z == -1`. С `$z` это произошло потому, что унарный минус считается операцией, поэтому перед изменением знака значение `1.8` усекается до `1`. Аналогично функции, предполагающие получение чисел с плавающей запятой, например `sqrt` или тригонометрические функции, продолжают принимать и возвращать вещественные числа даже в области действия `use integer`. Поэтому результат `sqrt(1.44)` равен `1.2`, но `0 + sqrt(1.44)` теперь просто `1`.

Используется арифметика целых чисел, предоставляемая компилятором С вашей платформы. Это означает, что собственная семантика Perl для арифметических операций может не сохраниться. Часто источником проблем служит остаток при делении отрицательных чисел. Perl может делать это одним способом, а ваша машина – другим:

```
% perl -le "print (4 % -3)"
-2

% perl -Minteger -le "print (4 % -3)"
1
```

Кроме того, целочисленная арифметика вынуждает поразрядные операторы считать свои операнды знаковыми целыми числами:

```
% perl -le "print ~0"
18446744073709551615

% perl -Minteger -le "print ~0"
-1
```

less

```
use less;

use less "CPU";
use less "memory";
use less "time";
use less "disk";
use less "fat";        # отлично сочетается с "use locale";
```

Реализованная в Perl версии v5.10 и более поздних, эта прагма по задумке должна советовать компилятору, генератору кода или интерпретатору идти на опреде-

ленные компромиссы, используя подсказки для новых ссылок на хеши, которые теперь возвращает caller.

Этот модуль всегда входил в состав дистрибутива Perl (как шутка), но ничего не делал до версии v5.10. Но даже теперь подсказки доступны только в лексической области видимости, поэтому less так и остается всего лишь демонстрацией возможностей новой функции caller, хотя из документации складывается впечатление, что другой модуль легко сможет выяснить, чего вам хотелось бы меньше (less).

Не будет ошибкой попросить Perl использовать чего-либо меньше (less), даже если он пока не умеет этого делать.

lib

```
use lib "$ENV{HOME}/libperl";  # добавить ~/libperl
no lib ".";                    # удалить cwd
```

Упрощает работу с @INC на этапе компиляции. Обычно применяется для добавления новых каталогов в маршрут поиска, чтобы позже команды do, require и use могли находить библиотечные файлы, которых нет в пути поиска Perl по умолчанию. Это особенно важно для use, потому что происходит и на этапе компиляции, а обычная установка @INC (т.е. на этапе выполнения) может произойти слишком поздно.

Параметры, передаваемые use lib, добавляются в начало маршрута поиска Perl. Сказать use lib *LIST* – это почти то же самое, что сказать BEGIN { unshift(@INC, *LIST*) }, но use lib *LIST* включает поддержку специфических для платформы каталогов. Для каждого каталога $dir из списка аргументов прагма lib проверяет также, существует ли каталог с именем *$dir/$archname/auto*. Если да, каталог *$dir/$archname* считается соответствующим специфическим для платформы каталогом и добавляется в @INC (перед $dir).

Во избежание избыточных записей, замедляющих поиск и расходующих память, дублирующиеся записи в конце @INC удаляются при добавлении.

Обычно требуется только добавлять каталоги в @INC. Если понадобится удалить каталоги из @INC, постарайтесь убрать только те, которые сами добавили, или те, в отношении которых вы уверены, что они не понадобятся другим модулям вашей программы. Другие модули могли добавить в @INC собственные каталоги, необходимые им для правильной работы.

Прагма no lib удаляет все вхождения указанных каталогов в @INC. Она также удаляет соответствующие специфические для платформы каталоги, как описано выше.

Загружаясь, прагма lib сохраняет текущее значение @INC в массив @lib::ORIG_INC, поэтому, чтобы восстановить исходное значение @INC, достаточно скопировать в нее этот массив.

Хотя @INC обычно содержит точку (.), текущий каталог – это не столь удобно, как может показаться. Во-первых, запись с точкой идет последней, а не первой, чтобы модули, установленные в текущем каталоге, не получали более высокий приоритет, чем их версии, уже установленные в системе. Если требуется иное поведение, можно сказать use lib ".". Хуже, что это текущий каталог процесса Perl, а не каталог, где располагается сценарий, что делает поиск в текущем каталоге совер-

шенно неоднозначным. Если мы создаем программу и несколько модулей, которые она должна использовать, все будет отлично во время разработки, но при запуске не из того каталога, где находятся файлы, все сломается.

Одно из решений заключается в применении стандартного модуля FindBin:

```
use FindBin,           # где проживает сценарий?
use lib $FindBin::Bin; # использовать этот каталог и для libs
```

Модуль FindBin пытается обнаружить полный путь к программе текущего процесса. Не используйте его для защиты, потому что злонамеренные программы, если постараются, смогут его обмануть. Но если не ломать модуль специально, он будет работать как нужно. Модуль предоставляет переменную $FindBin::Bin (которую можно импортировать), содержащую предположение модуля о каталоге установки программы. Можете использовать прагму lib, чтобы добавить этот каталог в свой массив @INC, создав путь, относящийся к выполняемому файлу.

Некоторые программы предполагают установку в каталог *bin* и пытаются обнаружить свои библиотечные модули в каталоге *lib*, на том же уровне, что и *bin*. Например, программы могут находиться в */usr/local/apache/bin* или */opt/perl/bin*, а библиотеки – в */usr/local/apache/lib* и */opt/perl/lib*. Следующий код ловко использует это обстоятельство:

```
use FindBin qw($Bin);
use lib "$Bin/../lib";
```

Если в нескольких разных программах заданы одни и те же параметры use lib, возможно, будет уместно установить переменную среды PERL5LIB. См. описание переменной среды PERL5LIB в главе 17.

```
# синтаксис для sh, bash, ksh или zsh
$ PERL5LIB=$HOME/perllib; export PERL5LIB

# синтаксис для csh или tcsh
% setenv PERL5LIB ~/perllib
```

Если необходимо работать с дополнительными каталогами только в этой программе, не меняя ее исходного кода, воспользуйтесь ключом командной строки *-I*:

```
% perl -I ~/perllib program-path args
```

Подробнее о применении *-I* в командной строке читайте в главе 17.

locale

```
@x = sort @y;        # сортировка в порядке ASCII
{
    use locale;
    @x = sort @y;    # использовать при сортировке национальные установки
}
@x = sort @y;        # снова сортировка в порядке ASCII
```

Эта прагма с лексической областью видимости говорит компилятору, что нужно включить (или выключить, если это прагма no locale) использование национальных настроек POSIX во встроенных операциях. При включенных настройках функции преобразования регистра и механизм поиска по шаблону учитывают

языковую среду, разрешая символы с диакритическими знаками и тому подобное. В области действия этой прагмы, если ваша библиотека С знакома с национальными настройками POSIX, Perl будет учитывать значение LC_CTYPE в регулярных выражениях и LC_COLLATE при сравнении строк, например в sort.

Поскольку эти установки являются скорее формой национализма, а не интернационализма, их применение может давать неожиданные эффекты при использовании Юникода. Более переносимый и надежный способ преобразования регистра и сравнения строк – использовать встроенную поддержку Юникода, стандартную для всех вариантов Perl, а не полагаться на, возможно, слишком замысловатые национальные настройки. За дополнительными подробностями обращайтесь к разделам «Сравнение и сортировка строк Юникода» и «Сортировка с учетом региональных настроек» в главе 6.

mro

```
use mro;        # включает next::method и родственные ему глобально

use mro "dfs"; # включает DFS MRO для данного класса (по умолчанию)
use mro "c3";  # включает C3 MRO для данного класса
```

По умолчанию поиск методов в Perl сначала выполняется в глубину, по классам (пакетам) в @INC. Прагма mro изменяет этот порядок поиска. С аргументом dfs (depth-first search) поиск по умолчанию ведется сначала в глубину, а с аргументом c3 используется алгоритм C3, позволяющий разрешать некоторые неоднозначности при множественном наследовании. В отсутствие списка импортирования сохраняется порядок поиска методов по умолчанию посредством включения некоторых возможностей, взаимодействующих с алгоритмом C3 (см. главу 12).

open

```
use open IN  => ":crlf", OUT => ":raw";
use open OUT => ":utf8";
use open IO  => ":encoding(iso-8859-7)";

use open IO  => ":locale";

use open ":encoding(utf8)";
use open ":locale";
use open ":encoding(iso-8859-7)"

use open ":std";
```

Объявляет один или несколько фильтров ввода/вывода (прежнее название – *дисциплины*), но только если Perl был собран с поддержкой PerlIO. Любой оператор open и readpipe (т.е. qx// или обратные апострофы), попадающий в область действия этой прагмы и не указавший собственные фильтры, использует эти объявления по умолчанию. В любом случае, эта прагма не оказывает влияния на open с явно заданным набором фильтров, равно как и на sysopen.

В настоящее время на выбор имеется несколько фильтров:

:bytes

Считает данные символами с кодами в диапазоне от 0 до 255. Противоположен фильтру :utf8. Отличается от фильтра :raw, поскольку все же может обрабатывать комбинации CRLF в Windows.

:crlf

Соответствует текстовому режиму, когда окончания строк преобразуются в родные для системы последовательности и обратно. Не выполняет никаких действий в системах, где binmode является пустой операцией. Доступен без поддержки PerlIO.

:encoding(*ENCODING*)

Определяет кодировку, поддерживаемую модулем Encode, прямо или косвенно.

:locale

Обеспечивает кодирование и декодирование данных в соответствии с национальными настройками.

:raw

Этот псевдофильтр отключает все нижележащие фильтры, которые не интерпретируют данные как двоичные. Не выполняет никаких действий в системах, где binmode является пустой операцией. Доступен без поддержки PerlIO.

:std

Фильтр :std в действительности не является фильтром. Он применяет другие фильтры, указанные в директиве, к стандартным дескрипторам файлов. С аргументом OUT устанавливает фильтры для стандартного вывода. С аргументом IN – для стандартного ввода.

:utf8

Обеспечивает кодирование и декодирование данных в кодировке UTF-8, интерпретируя их как строки символов. Противоположен фильтру :bytes.

При использовании встроенного фильтра utf8 в потоках ввода очень важно подготовить дескриптор файла к обработке ошибок кодирования. Вот как это сделать наилучшим образом:

```
use warnings FATAL => "utf8"; # на случай появления ошибок кодирования
```

При таком подходе ошибки кодирования будут вызывать исключения. Восстановление нормальной работы при наличии таких ошибок возможно, но сопряжено с большими сложностями.

ops

```
perl -M-ops=system .   # запретить код операции "system"
```

Прагма ops запрещает некоторые коды операций с необратимым глобальным эффектом. Перед запуском программы интерпретатор Perl всегда компилирует исходный код во внутреннее представление. По умолчанию на генерируемые коды операций не накладывается никаких ограничений. Запрещая коды операций, можно управлять компиляцией сценариев; попытка скомпилировать запрещенную операцию будет вызывать ошибку компиляции. Однако не следует думать,

что это позволяет повысить безопасность. Дополнительную информацию о кодах операций можно найти в описании модуля Opcode. Обратите также внимание на модуль Safe (глава 20), который может оказаться более удачным выбором.

overload

В модуле Number:

```
package Number;
use overload "+"  => \&myadd,
             "-"  => \&mysub,
             "*=" => "multiply_by";
```

В пользовательской программе:

```
use Number;
$a = Number->new( 57 );
$b = $a + 5;
```

Встроенные операторы хорошо работают со строками и числами, но имеют мало смысла в применении к ссылкам на объекты (поскольку, в отличие от C и C++, Perl не поддерживает арифметику указателей). Прагма overload позволяет переопределить алгоритмы работы этих встроенных операций для объектов, созданных программистом. В предыдущем примере вызов прагмы переопределяет три операции над объектами Number: сложение будет вызывать функцию Number::myadd, вычитание – функцию Number::mysub, а оператор присваивания с умножением – метод multiply_by в классе Number (или одном из его базовых классов). Мы говорим, что эти операторы теперь *перегружены*, потому что на них возложен дополнительный смысл (а не потому, что у них теперь слишком много значений, хотя бывает и так).

Значительно подробнее о перегрузке сказано в главе 13.

overloading

```
no overloading;
```

Это одна из немногих прагм, которые чаще используются для отключения, а не для включения чего-либо. В данном случае прагма отключает все перегруженные операции, возвращая им нормальное поведение до конца лексической области видимости.

Чтобы отключить перегруженные операции, следует использовать те же ключи, что и в прагме overload:

```
no overloading qw(""); # отключить перегрузку преобразования в строку
```

Чтобы вновь включить перегрузку, выполните обратное действие:

```
use overloading;       # включить все обратно

use overloading @ops;  # включить только часть
```

parent

```
use parent qw(Mother Father);
```

Прагма parent служит заменой прагмы base. Она загружает модули и устанавливает отношения наследования без обращения к волшебству хеша %FIELDS. Поддерживает возможность создания иерархии наследования без загрузки файлов.

Следующий пример эквивалентен загрузке обоих родительских модулей и добавлению их в @INC без явного объявления @INC:

```
BEGIN {
    require Mother;
    require Father;
    push @ISA  qw(Mother Father):
}
```

Предполагается, что каждый родительский модуль находится в отдельном файле. Если родительские классы определены в том же файле или уже были загружены из файла как часть другого класса, можно воспользоваться параметром -norequire, который просто устанавливает отношения наследования:

```
use parent qw(-norequire Mother Father);
```

Эта строка эквивалентна добавлению указанных классов в @ISA:

```
BEGIN {
    push @ISA, qw(Mother Father):
}
```

re

Управляет применением регулярных выражений. Может вызываться пятью способами: taint, eval и /flags, имеющими лексическую область видимости, а также debug и debugcolor, которые ее не имеют.

```
use re "taint";
# Содержимое $match меченое, если $dirty тоже меченая.
($match) = ($dirty =~ /^(.*)$/s);

# Разрешить интерполяцию кода:
use re "eval";
$pat = '(?{ $var = 1 })'; # выполнение встроенного кода
/alpha${pat}omega/;       # даст ошибку при -T
                          # и мечености $pat


use re "/a";              # по умолчанию все шаблоны получат флаг /a
use re "/msx";            # по умолчанию все шаблоны получат флаги /msx

use re "debug";           # как "perl -Dr"
/^( *)$/s;                # выводить отладочную информацию
                          # на этапах компиляции и выполнения

use re "debugcolor":      # то же. что "debug", но с расцветкой вывода

use re qw(Debug LIST);    # точное управление выводом отладочной информации
```

В области действия use re "taint", когда объект регулярного выражения является меченой строкой, нумерованные переменные регулярного выражения и значения, возвращаемые оператором m// в списочном контексте, являются мечеными. Это полезно, когда применение регулярных выражений к меченым данным направлено не на извлечение безопасных подстрок, а на осуществление других преобразований. Читайте описание меченых данных в главе 20.

Если действует use re "eval", в регулярном выражении разрешены утверждения, выполняющие код Perl и имеющие вид (?{ }), даже если регулярное выражение содержит интерполируемые переменные. Выполнение фрагментов кода в результате интерполяции переменных в регулярное выражение обычно запрещено по причинам безопасности: нехорошо, если программы, читающие шаблоны из файлов конфигурации, аргументы командной строки или поля форм CGI, вдруг начнут выполнять произвольный код, если только не были специально разработаны для такой цели. Эта прагма разрешает интерполяцию только немеченых строк; меченые данные все равно будут возбуждать исключения (если выполнение производится при включенной проверке меченых данных). См. также главы 5 и 20.

Для этой прагмы интерполяция предварительно скомпилированных регулярных выражений (создаваемых оператором qr//) не считается интерполяцией переменных. Тем не менее при создании шаблона qr// требуется, чтобы действовала use re "eval", если в интерполированных строках содержатся утверждения с кодом. Например:

```
$code = '(?{ $n++ })';     # кодовое утверждение
$str = '\b\w+\b'   $code;  # сконструировать интерполируемую строку

$line =~ /$str/;           # здесь нужна use re "eval"

$pat = qr/$str/;           # здесь тоже нужна use re "eval"
$line =~ /$pat/;           # а здесь не нужна use re "eval"
```

Режим флагов, use re "/flags", включает модификаторы шаблонов по умолчанию для операций поиска, подстановки и для операторов заключения регулярных выражений в кавычки в лексической области действия прагмы. Например, если необходимо, чтобы все шаблоны в файле использовали семантику ASCII для символьных классов (\d, \w и \s):

```
while (<>) {
    use re "/a";
    if (/\d/) {    # только 0 .. 9
        print "Найдена цифра ASCII: $_";
    }
}
```

Включение одного из модификаторов шаблонов, воздействующих на символьные классы, как традиционные, так и POSIX (/adlu), переопределяет любые настройки, выполненные с помощью прагмы locale или возможности unicode_strings.

Чтобы включить поддержку многострочного режима, когда ^ и $ соответствуют символам перевода строки, а не только началу и концу текста (/m), точка (.) соответствует символу перевода строки (/s) и применяется расширенный режим определения шаблонов (/x), используйте:

```
use re "/msx";
```

В области действия use re "debug" Perl выводит отладочные сообщения во время компиляции и выполнения регулярных выражений. Вывод получается такой же, как при выполнении «отладочного Perl» (скомпилированного с ключом -DDEBUGGING компилятора C) и выполнении программы Perl с ключом *-Dr* командной строки Perl. В зависимости от сложности шаблона вывод может получиться огромным. Вызов use re "debugcolor" раскрашивает вывод, что может быть удобно, если терминал понимает последовательности, управляющие цветом. Назначьте переменной среды PERL_RE_TC в качестве значения список свойств *termcap*(5) для выделения, разделив значения запятыми. Дополнительные сведения вы найдете в главе 18.

Чтобы получить более полное управление отладочным выводом, используйте Debug (с большой буквы D) и список того, что требует отладки. Параметр All эквивалентен использованию use re "debug":

```
{
    use re qw(Debug All); # то же, что и "use re 'debug'"
    ...
}
```

Для прицельного управления отладочным выводом пробуйте другие варианты. Например, параметр COMPILE обеспечит отладку только инструкций, связанных с компиляцией шаблона:

```
{
    use re qw(Debug COMPILE); # то же, что и  use re "debug"
    ...;
}
```

Другие возможные параметры перечислены в документации к модулю re.

sigtrap

```
use sigtrap;
use sigtrap qw(stack-trace old-interface-signals); # то же самое

use sigtrap qw(BUS SEGV PIPE ABRT);
use sigtrap qw(die INT QUIT);
use sigtrap qw(die normal-signals);
use sigtrap qw(die untrapped normal-signals);
use sigtrap qw(die untrapped normal-signals
               stack-trace any error-signals);

use sigtrap "handler" => \&my_handler, "normal-signals";
use sigtrap qw(handler my_handler normal-signals stack-trace error-signals);
```

Устанавливает некоторые простые обработчики сигналов, чтобы программисту не нужно было о них беспокоиться. Это полезно, если неперехваченный сигнал может вызвать нарушение работы программы, как бывает, если программа содержит блоки END {}, вызывает деструкторы объектов или другую обработку на выходе, которую нужно произвести независимо от способа завершения работы программы.

Когда выполнение программы прерывается из-за неперехваченного сигнала, она немедленно завершает работу без выполнения заключительных операций. Если программист предусмотрит обработку и преобразование сигналов в фатальные

исключения, при их появлении будет происходить следующее: будет выполнен выход из всех областей видимости с освобождением занятых в них ресурсов и будет произведена обработка всех блоков END.

Прагма sigtrap предоставляет два простых обработчика сигналов. Один обеспечивает трассировку стека Perl, а другой возбуждает обычное исключение посредством die. Можно также передать прагме собственный обработчик. Можно задать предопределенные группы перехватываемых сигналов или передать собственный список сигналов явным образом. Кроме того, прагма способна установить обработчики только для тех сигналов, которые не обрабатываются иным способом.

Аргументы, передаваемые в use sigtrap, обрабатываются по порядку. Когда встречается определенное пользователем имя сигнала или имя одного из предопределенных списков сигналов, немедленно устанавливается обработчик. Когда встречается ключ прагмы, его действие распространяется только на обработчики, устанавливаемые далее при обработке списка аргументов.

Обработчики сигналов

Следующие ключи определяют, какой обработчик будет использоваться для сигналов, расположенных далее в списке:

stack-trace

Этот обработчик, предоставляемый прагмой, выводит трассировку стека Perl в STDERR, а затем пытается выполнить дамп памяти. Это обработчик сигналов по умолчанию.

die

Этот обработчик, предоставляемый прагмой, вызывает die через Carp::croak, передавая сообщение, указывающее на перехваченный сигнал.

handler *YOURHANDLER*

YOURHANDLER будет играть роль обработчика сигналов для встретившихся далее в списке сигналов. *YOURHANDLER* может быть любым значением, допустимым для %SIG. Помните, что внутри обработчика сигнала нельзя гарантировать правильную работу многих библиотечных вызовов C (особенно для стандартного ввода/вывода). Хуже того, сложно определить, какой код библиотеки C каким кодом Perl вызывается. (С другой стороны, многие из сигналов, которые перехватывает sigtrap, довольно жестоки – они в любом случае уничтожат программу, так что нет особого вреда в том, чтобы *попытаться* что-то сделать.)

Предопределенные списки сигналов

Прагма sigtrap имеет несколько встроенных списков сигналов:

normal-signals

Сигналы, с которыми обычно может столкнуться программа и которые по умолчанию вызывают ее завершение. Это сигналы HUP, INT, PIPE и TERM.

error-signals

Сигналы, которые обычно указывают на серьезные проблемы в интерпретаторе Perl или в программе. А именно ABRT, BUS, EMT, FPE, ILL, QUIT, SEGV, SYS и TRAP.

old-interface-signals

Эти сигналы по умолчанию перехватывались в старой версии интерфейса sigtrap. Это сигналы ABRT, BUS, EMT, FPE, ILL, PIPE, QUIT, SEGV, SYS, TERM и TRAP. Если в use sigtrap не передать конкретные сигналы или списки, используется именно этот список.

Если на данной платформе не реализован некоторый сигнал, указанный в предопределенных списках, имя этого сигнала просто игнорируется. (Сам сигнал нельзя проигнорировать, поскольку он не существует.)

Прочие аргументы sigtrap

untrapped

Подавляет установку обработчиков для сигналов, перечисляемых далее в списке, если они уже перехватываются или игнорируются.

any

Устанавливает обработчики для всех сигналов, перечисляемых далее в списке. Это режим по умолчанию.

signal

Любой аргумент, который выглядит как имя сигнала (т.е. соответствует шаблону /^[A-Z][A-Z0-9]*$/), запрашивает обработку этого сигнала прагмой sigtrap.

number

Числовой аргумент требует, чтобы номер версии прагмы sigtrap был не меньше number. Действует так же, как в обычных модулях, имеющих переменную пакета $VERSION:

```
% perl -Msigtrap -le 'print $sigtrap::VERSION'
1.02
```

Примеры использования sigtrap

Обеспечить трассировку стека для сигналов старого интерфейса:

```
use sigtrap;
```

То же самое, но более явно:

```
use sigtrap qw(stack-trace old-interface-signals);
```

Обеспечить трассировку стека только для четырех перечисленных сигналов:

```
use sigtrap qw(BUS SEGV PIPE ABRT);
```

Выполнить die при получении сигнала INT или QUIT:

```
use sigtrap qw(die INT QUIT);
```

Выполнить die при получении любого из сигналов: HUP, INT, PIPE или TERM:

```
use sigtrap qw(die normal-signals);
```

Выполнить die при получении любого из сигналов: HUP, INT, PIPE или TERM – но не изменяя режим для сигналов, которые уже перехватываются или игнорируются в каком-либо месте программы:

```
use sigtrap qw(die untrapped normal-signals);
```

Выполнить die **при получении любого из не перехватываемых в данный момент сигналов из списка** normal-signals**; кроме того, обеспечить вывод трассировки стека при получении любого из сигналов** error-signals**:**

```
use sigtrap qw(die untrapped normal-signals
               stack-trace any error-signals);
```

Установить подпрограмму my_handler **в качестве обработчика для** normal-signals**:**

```
use sigtrap "handler" => \&my_handler, "normal-signals";
```

Установить my_handler **в качестве обработчика для** normal-signals**; обеспечить вывод трассировки стека при получении любого из сигналов** error-signals**:**

```
use sigtrap qw(handler my_handler normal-signals
              stack-trace error-signals);
```

sort

До версии v5.8 встроенная функция sort **использовала алгоритм *быстрой сортировки* (*quicksort*). Этот алгоритм в худших случаях показывал квадратичную сложность и мог не сохранять первоначальный порядок следования элементов с одинаковыми значениями (т.е. проявлял *нестабильность*). Начиная с Perl v5.8 по умолчанию применяется алгоритм *сортировки слиянием* (*mergesort*), который в худшем случае показывает производительность *O(n log n)* и сохраняет первоначальный порядок следования элементов с одинаковыми значениями (т.е. проявляет *стабильность*).**

Прагма sort **позволяет выбирать используемый алгоритм сортировки. И если когда-нибудь по умолчанию будет использоваться другой алгоритм, отличный от алгоритма сортировки слиянием, вы сможете выбрать стабильную сортировку, не указывая конкретный алгоритм:**

```
use sort "stable";     # гарантирует стабильность
use sort "_quicksort"; # использовать алгоритм быстрой сортировки
use sort "_mergesort"; # использовать алгоритм сортировки слиянием
use sort "defaults";   # вернуться к поведению по умолчанию
no sort "stable";      # стабильность не имеет значения

use sort "_qsort"      # синоним _quicksort

my $current,
BEGIN {
    $current = sort::current(); # идентифицировать господствующий алгоритм
}
```

strict

```
use strict;         # Установить все три структуры

use strict "vars"; # Переменные должны быть предварительно объявлены
use strict "refs"; # Нельзя использовать символические ссылки.
```

```
use strict "subs"; # Строки голых слов необходимо заключать в кавычки

use strict;        # Установить все...
no strict "vars";  # ...затем отменить одну.

use v5.12;         # по умолчанию для v5.2.0 или выше
```

Эта прагма с лексической областью видимости изменяет некоторые основные правила, согласно которым Perl определяет, что является допустимым кодом. Иногда эти ограничения кажутся слишком строгими для простых задач, например, когда мы просто пытаемся «слепить» программу-фильтр в пять строк. Чем больше программа, тем больше строгости необходимо проявлять по отношению к ней.

В настоящее время строгости целесообразны по отношению к трем вещам: subs, vars и refs. Если список импорта не задан, предполагается установка всех трех ограничений.

strict "refs"

Эта прагма генерирует ошибки этапа выполнения в случае применения символических ссылок, намеренного или случайного.

Подробнее о них читайте в главе 8.

```
use strict "refs";

$ref = \$foo;      # Сохранить "настоящую" (жесткую) ссылку.
print $$ref;       # Разыменование допускается.

$ref = "foo";      # Сохранение имени глобальной (пакетной) переменной.
print $$ref;       # НЕВЕРНО, ошибка этапа выполнения при strict refs.
```

Символические ссылки подозрительны по ряду причин. Даже добропорядочные программисты могут на удивление легко допустить ошибочное их применение; прагма strict "refs" предохраняет от этого. В отличие от настоящих, символические ссылки могут относиться только к глобальным переменным. Для них не ведется учет числа ссылок. Часто есть способ лучше для решения этой задачи: вместо ссылки на имя в глобальной таблице имен используйте хеш как самостоятельную мини-таблицу имен. Это более эффективно, лучше читается и оставляет меньше возможностей ошибиться.

Тем не менее некоторые виды допустимых действий в действительности требуют прямого доступа к таблице глобальных символов пакета для имен переменных и функций. Например, может потребоваться исследовать список @EXPORT или надкласс @ISA некоторого пакета, имя которого заранее не известно. Либо может понадобиться установить массу вызовов функций, каждый из которых является псевдонимом одного и того же замыкания. Это как раз то, для чего символические ссылки хороши, но, чтобы использовать их в области действия use strict, необходимо предварительно снять ограничение refs:

```
# создать группу методов доступа к атрибуту
for my $methname (qw/name rank serno/) {
    no strict "refs";
    *$methname = sub { $_[0]->{ __PACKAGE__ . $methname } };
}
```

strict "vars"

При этом ограничении ошибка этапа компиляции возникает всякий раз при попытке доступа к переменной, которая не удовлетворяет хотя бы одному из следующих критериев:

- Предопределена в самом Perl, например @ARGV, %ENV и глобальные переменные с именами из знаков пунктуации, такие как $. или $_.
- Объявлена через our (для глобальной) или my/state (для лексической).
- Импортирована из другого пакета. (Прагма vars подделывает импорт, но можно применить our.)
- Имеет абсолютное имя, включающее имя пакета и два двоеточия в качестве разделителя имени.

Одного только оператора local недостаточно, чтобы удовлетворить use strict "vars", поскольку, несмотря на свое имя, этот оператор не изменяет глобальную область видимости переменной. Он просто дает переменной новое временное значение до конца блока на этапе выполнения. Для объявления глобальной переменной по-прежнему необходима директива our, а для объявления лексической переменной – my или state. Однако our можно локализовать:

```
local our $law = "martial";
```

На глобальные переменные, предопределенные в Perl, эти требования не распространяются. Это относится к переменным, глобальным для всей программы (которые помещены в пакет main, подобно @ARGV или $_) и к переменным пакета, таким как $a и $b, обычно используемым функцией sort. Пакетные переменные, используемые модулями типа Exporter, все же нуждаются в объявлении через our:

```
our @EXPORT_OK = qw(name rank serno);
```

strict "subs"

Данное ограничение предписывает Perl считать все голые слова ошибками синтаксиса. *Голое слово* («bareword» или, в некоторых диалектах, «bearword») является любым «голым» именем или идентификатором, для которого нет иной интерпретации, продиктованной контекстом. (Контекст часто диктуется соседним ключевым словом или лексемой либо предварительным объявлением рассматриваемого слова.) Традиционно голые слова интерпретировались как строки без кавычек. Данное ограничение делает такую интерпретацию незаконной. Если вы намерены работать с именем как со строкой, заключите его в кавычки. Чтобы использовать имя как вызов функции, необходимо предварительное объявление или применение круглых скобок.

Особым случаем диктуемого контекста является появление голого слова в фигурных скобках, слева от оператора =>, которое рассматривается как заключенное в кавычки и потому не подпадает под данное ограничение.

```
use strict "subs";

$x = whatever;                    # НЕВЕРНО: ошибка "голого слова"!
$x = whatever();                  # А вот это всегда работает.
```

```
sub whatever;                    # Предварительное объявление функции.
$x = whatever;                   # Теперь порядок.

# Такое применение разрешено, поскольку => заключает в кавычки
%hash = (red => 1, blue => 2, green => 3);

$rednum = $hash{red};            # Ok, фигурные скобки здесь закавычивают

# А вот здесь нет:
@coolnums = @hash{blue, green};  # НЕВЕРНО: ошибка "голого слова".
@coolnums = @hash{"blue", "green"}, # Порядок, слова теперь в кавычках.
@coolnums = @hash{qw/blue green/};  # Аналогично
```

subs

```
use subs qw/winken blinken nod/,
@x = winken 3..10;
@x = nod blinken @x
```

Объявляет все имена в списке аргументов как стандартные подпрограммы. Преимущество в том, что затем можно использовать эти функции без круглых скобок как списочные операторы, как если бы они были объявлены самим программистом. Это не всегда так же полезно, как полное объявление, поскольку делает невозможным использование прототипов или атрибутов, таких как:

```
sub winken(@);
sub blinken(\@) : locked;
sub nod($) : lvalue;
```

Основанная на стандартном механизме импорта, прагма `use subs` имеет не лексическую область видимости, а область видимости пакета. Это значит, что объявления действуют во всем файле, где они упомянуты, но только в текущем пакете. Такие объявления нельзя отменить с помощью `no subs`.

threads

В настоящее время Perl поддерживает две модели организации многопоточного выполнения. Одна из них – старая модель, реализованная в Perl версии v5.005 посредством модуля `Threads`, который был исключен в Perl версии v5.10. Вторая модель появилась в Perl версии v5.8 и называется «потоки интерпретатора» (interpreter threads, или ithreads) – она запускает отдельный экземпляр интерпретатора Perl для каждого потока выполнения. Если вы знакомы с многопоточными моделями в других языках программирования, забудьте о них – они не имеют ничего общего с многопоточной моделью в Perl, кроме названия.

Для использования прагмы `threads` необходимо, чтобы интерпретатор *perl* был собран с поддержкой потоков выполнения. Чтобы выяснить, так ли это, выполните команду `perl -V` и поищите в строках вывода нечто похожее на `USE_ITHREADS` в параметрах компилятора. Можно также воспользоваться модулем `Config`, который позволяет получить параметры компиляции во время выполнения вашей программы:

```
use Config;
$Config{useithreads}
or die("Пересоберите Perl с поддержкой потоков, чтобы выполнить эту программу.");
```

Многие дистрибутивы Perl, распространяемые в составе операционных систем, уже обладают поддержкой многопоточной модели выполнения, т. к. проще включить эту поддержку для всех, чем выжать дополнительную производительность, отключив ее для всех, и выслушивать жалобы на отсутствие этой поддержки.

Ниже приведен короткий пример. Он запускает несколько потоков выполнения, обособляет их, затем запускает заключительный поток и присоединяет его. Программа не ждет завершения обособленных потоков выполнения, но она будет ждать завершения присоединенного потока. Потоки можно создавать с применением ссылок на код или имен подпрограмм, а можно воспользоваться функцией async из прагмы threads:

```
#!/usr/bin/perl
use v5.10;

use Config;
$Config{useithreads}  || die "Требуется поддержка потоков для запуска";

use threads;

threads->create(sub {
    my $id = threads->tid;
    foreach (0 .. 10) {
        sleep rand 5;
        say "Мяу от кошки $id ($_)";
    }
})->detach;

for (0 .. 4) {
    my $t = async {
        my $id = threads->tid;
        foreach (0 .. 10) {
            sleep rand 5;
            say "Гав от собаки $id ($_)";
        }
    };
    $t->detach;
    return $t;
};

threads->create("bird")->join;
sub bird {
    my $id = threads->tid;
    for (0 .. 10) {
        sleep rand 5;
        say "Чирик от птички $id ($_)"
    }
}
```

Дополнительную информацию о потоках выполнения можно получить на странице *perlthrtut* – учебнике по использованию потоков в Perl. В Perl предусмотрена

возможность совместного использования переменных в потоках выполнения посредством thread::shared и общей очереди с помощью модуля Threads::Queue.

utf8

```
use utf8;
```

Прагма utf8 объявляет, что исходный текст на языке Perl до конца лексической области имеет кодировку UTF-8. Это позволяет использовать идентификаторы и строки, содержащие символы Юникода.

```
use utf8;
my $résumé_name = "Björk Guðmundsdóttir"
{
    no utf8;
    my $mojibake = '文字化け'   # вероятно  вызовет ошибку
}
```

Прагма utf8 предоставляет также некоторые другие возможности, но в этом отношении мы рекомендуем отдавать предпочтение модулю Encode.

Обратите внимание, что, начиная с Perl версии v5.14, компилятор не нормализует идентификаторы, поэтому вы не сможете различать глифы, сформированные разными способами (с использованием композиционных или декомпозиционных символов). Подробнее о нормализации рассказывается в главе 6. Мы рекомендуем нормализовать все идентификаторы в форму NFC (или NFKC), чтобы избежать вероятности появления разных переменных, имена которых выглядят одинаково.

vars

```
use vars qw($frobbed @munge %seen):
```

Эта прагма, когда-то применявшаяся для объявления глобальных переменных, в настоящее время уступает место модификатору our. Предшествующее объявление лучше записать так:

```
our($frobbed, @munge, %seen);
```

или даже:

```
our $frobbed = "F";
our @munge = "A"     $frobbed;
our %seen = ();
```

При любой форме записи помните, что прагма our объявляет глобальные переменные пакетов, а не лексические переменные файлов.

version

```
use version 0.77;

my $version = version->parse($version_string),
my $qversion = qv($other_version_string);
```

```
if ($version > $qversion) {
    say "Version is greater!";
}
```

В действительности модуль version прагмой не является, но выглядит как прагма, потому что его имя состоит только из символов нижнего регистра. До версии v5.10 модуль version предоставлял способ заключать версии в кавычки с помощью оператора qv() и способ сравнивать номера версий. Может показаться, что в этом нет ничего сложного, но стоит погрузиться в решение этой задачи, можно даже усомниться в своих способностях к программированию. Например, в каком порядке выходили версии 1.02, 1.2 и v1.2.0? Теперь Perl способен решать эту задачу самостоятельно. Однако от этого она не стала проще[1].

vmsish

```
use vmsish;             # все возможности

use vmsish "exit";
use vmsish "hushed";
use vmsish "status";
use vmsish "time";

no vmsish "hushed";
vmsish::hushed($hush);

use vmsish;             # все возможности
no vmsish "time";       # отключить 'time
```

Прагма vmsish управляет некоторыми возможностями Perl, ориентированными на VMS, чтобы ваши программы меньше напоминали программы для UNIX и больше – для VMS. Эти возможности имеют лексическую область видимости, благодаря чему вы можете включать и отключать их по мере необходимости.

exit

При включенной возможности exit оба вызова, exit 1 и exit 0, отображаются в значение SYS$NORMAL, свидетельствующее об успешном завершении. В интерпретации UNIX вызов exit 1 говорит об ошибке.

hushed

Если включить hushed, программа на Perl, запущенная из DCL, не выводит сообщения в SYS$OUTPUT или SYS$ERROR, завершаясь с ошибкой. Однако эта прагма не подавляет вывод сообщений от самой программы. Она воздействует только на функции exit и die в лексической области видимости, и только на те, что будут скомпилированы после того, как встретится эта прагма.

---

[1] Вам определенно понравится статья Дэвида Голдена (David Golden) «Version numbers should be boring» (*http://www.dagolden.com/index.php/369/version-numbers-should-be-boring/*).

status

Если включить status, в качестве возвращаемого значения функции system и значения переменной $? используется код завершения VMS, а не код завершения POSIX.

time

После включения этой возможности все временные значения будут откладываться относительно локального часового пояса, а не относительно Universal Time, используемого по умолчанию.

warnings

```
use warnings;       # то же, что импорт "всех"
no warnings;        # то же, что отмена импорта "всех"

use warnings::register;
if (warnings::enabled()) {
    warnings::warn("some warning")
}

if (warnings::enabled("void")) {
    warnings::warn("void", "some warning");
}

warnings::warnif("Warnings are on");
warnings::warnif("number", "Something is wrong with a number");
```

Эта прагма с лексической областью видимости допускает гибкое управление встроенными предупреждениями Perl, выдаваемыми как компилятором, так и системой времени выполнения.

Когда-то управлять обработкой предупреждений в программе Perl можно было только с помощью параметра командной строки *-w* или переменной $^W. Это полезно, но это подход «все или ничего». Применение ключа *-w* приводит к выводу предупреждений для кода модулей, которые могли быть написаны кем-то другим, что может озадачивать вас и смущать первоначального автора. Применение $^W для включения или выключения сообщений в блоках кода может оказаться не очень удачным, поскольку действует только на этапе выполнения, а не во время компиляции.[1] Другая проблема состоит в том, что эта глобальная для всей программы переменная имеет динамическую, а не лексическую область видимости. Это означает, что если мы включаем ее в блоке, а затем вызываем из него другой код, то снова рискуем включить вывод предупреждений в коде, который разрабатывался без следования точным стандартам.

Прагма warnings обходит эти ограничения, поскольку является механизмом этапа компиляции с лексической видимостью, позволяющим осуществлять тонкий контроль над тем, где следует, а где не следует – выводить предупреждения. Определена целая иерархия категорий предупреждений (рис. 29.1), обеспечивающая

---

[1] Если, конечно, нет блоков BEGIN.

независимое включение и выключение групп предупреждений. (Деление на категории является экспериментальным, так что может измениться.) Эти категории можно объединять, передавая в use **или** no **несколько аргументов:**

```
use warnings qw(void redefine),
no warnings qw(io syntax untie);
```

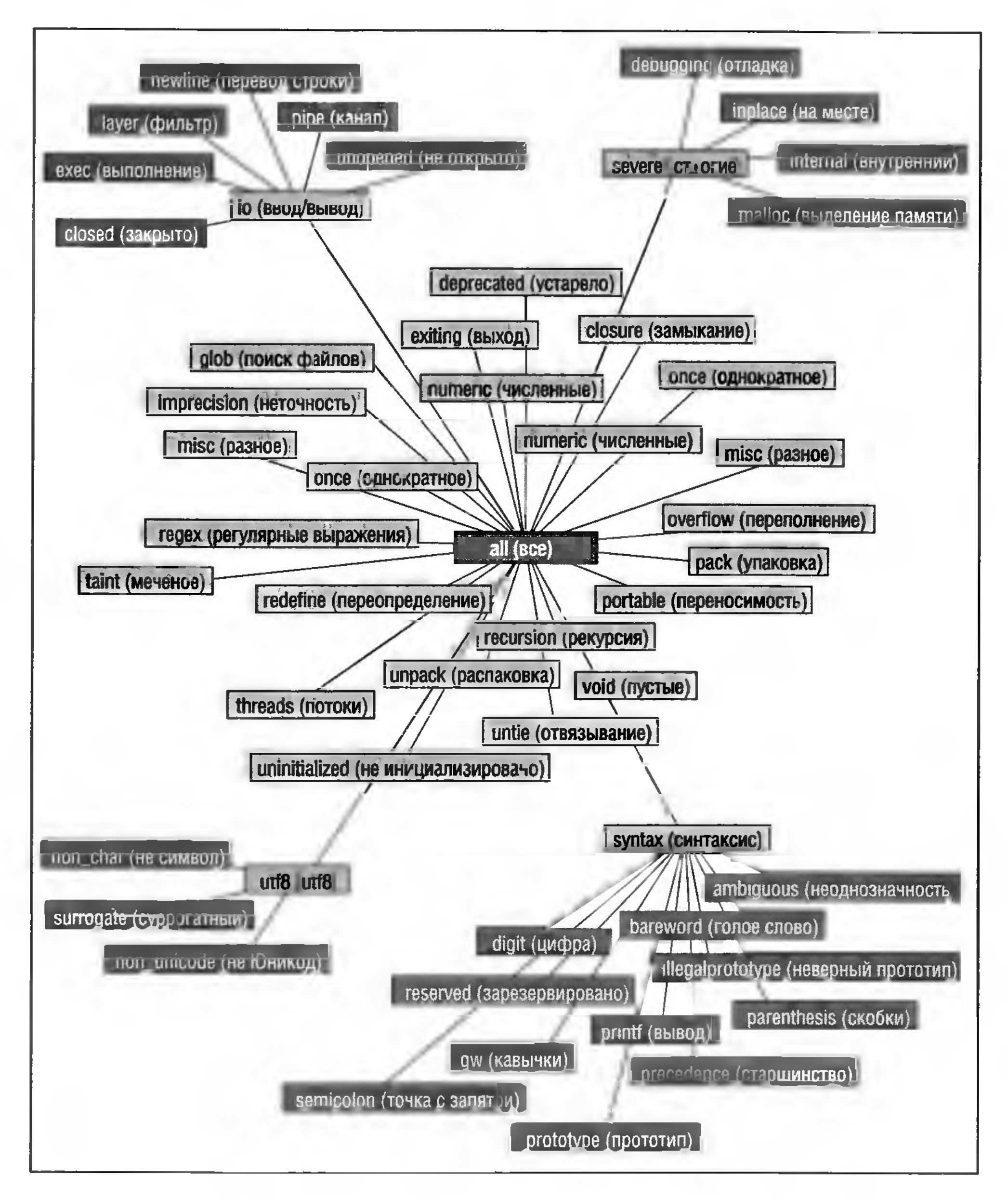


Рис. 29.1. Категории предупреждений Perl

Если в области видимости несколько экземпляров прагмы warnings, их действие суммируется:

```
use warnings "void";   # Включена только категория "void".
...
use warnings "io";     # Включены категории "void" и "io".
...
no warnings "void";    # Включена только категория "io"
```

Чтобы превратить предупреждения, включаемые некоторой прагмой warnings, в фатальные ошибки, предварите список импорта словом FATAL. Это удобно, если требуется, чтобы некоторое условие, обычно вызывающее предупреждение, прерывало выполнение программы. Допустим, например, что мы считаем использование неподходящей строки в качестве числа (которое обычно в этом случае принимает нулевое значение) совершенно недопустимым и хотим, чтобы это возмутительное событие прерывало программу. Попутно мы решили, что использование неинициализированных значений там, где ожидаются действительные строки или числа, тоже должно приводить программу к немедленному самоубийству:

```
{
    use warnings FATAL => qw(numeric uninitialized);
    $x = $y + $z;
}
```

Теперь, если $y или $z не инициализированы (т.е. содержат особое скалярное значение undef) или содержат строки, которые не очень хорошо преобразуются в числа, то вместо того, чтобы безмятежно двигаться дальше или, самое худшее, слегка пожаловаться, если включен вывод предупреждений, наша программа возбудит исключение. (Представьте себе Perl, работающий в режиме Python.) Если не обеспечить перехват исключительных ситуаций, эта ошибка станет фатальной. Текст исключительной ситуации тот же, что появился бы в обычном предупреждении.

Сделать фатальными все предупреждения, сказав в начале программы:

```
use warnings FATAL => "all";
```

не самое лучшее решение, потому что прагма не различает предупреждения этапов компиляции и выполнения. Первое же сообщение компилятора наверняка разойдется с вашими ожиданиями, но, если все предупреждения объявить фатальными, только это сообщение вы и увидите. Лучше отложить превращение предупреждений в фатальные ситуации до этапа выполнения.

```
use Carp qw(carp croak confess cluck);
use warnings;      # предупреждения этапа компиляции

# во время выполнения, перед всем остальным
$SIG{__WARN__} = sub { confess "FATALIZED WARNING: @_" }
```

Альтернативное решение заключается в использовании cluck вместо confess. В этом случае точно так же будет выводиться дамп стека, но программа будет продолжать работать. Это может пригодиться при поиске в коде путей, ведущих к появлению предупреждений. Читайте описание хеша %SIG в главе 25, где приводятся дополнительные примеры.

Прагма warnings игнорирует ключ командной строки *-w* и значение переменной $^W; настройки самой прагмы имеют более высокий приоритет. Однако ключ командной строки *-W* отменяет прагму, включая полный вывод предупреждений во всем коде программы, даже в том, который загружается с помощью do, require или use. Иными словами, с ключом *-W* интерпретатор Perl действует так, как если бы в каждом блоке программы присутствовала прагма use warnings "all".

Существует несколько вспомогательных функций для тех авторов модулей, которые хотят, чтобы их функции в модулях вели себя как встроенные функции в отношении лексической области видимости вызвавшего их модуля (другими словами, чтобы пользователи модуля имели возможность лексически включать и отключать предупреждения, которые может выводить модуль):

warnings::register

Регистрирует имя текущего модуля как новую категорию предупреждений, чтобы пользователи модуля могли выключать поступающие от него предупреждения.

warnings::enabled(*CATEGORY*)

Возвращает истинное значение, если категория предупреждений *CATEGORY* включена в лексической области видимости вызывающего модуля. В противном случае возвращает ложное значение. Если *CATEGORY* не указана, используется имя текущего пакета.

warnings::warn(*CATEGORY*, *MESSAGE*)

Если вызывающий модуль не устанавливает *CATEGORY* как FATAL, функция выводит *MESSAGE* в STDERR. Если *CATEGORY* установлена как FATAL, выводит *MESSAGE* в STDERR, после чего программа завершается. Если *CATEGORY* не указана, используется имя текущего пакета.

warnings::warnif(*CATEGORY*, *MESSAGE*)

Действует подобно функции warnings::warn, но только если категория *CATEGORY* включена.

Пользовательские прагмы

В Perl v5.10 появилась возможность создавать собственные прагмы с лексической областью видимости. Хеш %^H содержит информацию, которая может использоваться другим программным кодом для получения подсказок о том, что вы собираетесь сделать, а функция caller принимает ссылку на версию этого хеша, действительную для запрашиваемого уровня:

```
my $hints = ( caller(1) )[10];
```

Это простой хеш с простыми значениями. Если не вдаваться в подробности, этот хеш может совместно использоваться несколькими потоками выполнения. Он устроен так, что исключает возможность хранения любых значений, кроме целых чисел, строк и undef. Это не является большим ограничением, так как в действительности вам достаточно лишь знать, включена или выключена та или иная возможность. Кроме того, данный хеш имеет лексическую область видимости, поэтому каждая лексическая область получает собственную версию хеша.

Чтобы создать собственную прагму, определите три подпрограммы: import, unimport и in_effect. Первые две вызываются неявно, директивами use и no. Обычно директива use включает возможность вызовом функции import, тогда как no отключает эту возможность, вызывая функцию unimport. Помимо специальных операций, функции import и unimport устанавливают флаг в %^H. Другой программный код, находящийся за пределами прагмы, может вызывать in_effect, чтобы определить, действует ли прагма, а функция может обращаться к хешу %^H, чтобы получить установленное вами значение.

Нет каких-то строгих правил, регламентирующих, что можно добавлять в хеш %^H, но не забывайте, что другие прагмы тоже пользуются этим хешем для выполнения своей работы, поэтому выбирайте ключи, которые наверняка не будут использоваться другими прагмами, как, например, имя вашего пакета.

Ниже представлен короткий пример прагмы, замещающей встроенную функцию sqrt другой, способной обрабатывать отрицательные числа (грубо). Директива use complex вызовет метод import, который добавит в %^H ключ complex со значением 1 и создаст подпрограмму с именем sqrt, использующую то же определение, что и complex::complex_sqrt. Функция complex_sqrt вызывает метод in_effect, чтобы определить, допускается ли извлекать корень из отрицательных чисел. В этом случае она извлекает квадратный корень из абсолютного значения и, если исходное значение меньше 0, добавляет "i" к результату:

```
use utf8;
use v5.10;

package complex;
use strict;
use warnings;
use Carp;

sub complex_sqrt {
    my $number = shift;
    if (complex::in_effect()) {
        my $root = CORE::sqrt(abs($number));
        $root .= "i" if $number < 0;
        return $root;
    }
    else {
        croak("Невозможно выполнить sqrt для $number") if $number < 0;
        CORE::sqrt($number)
    }
}

sub import {
    $^H{complex} = 1;
    my($package) = (caller(1))[0];
    no strict "refs";
    *{ "${package}::sqrt" } = \&complex::complex_sqrt;
}

sub unimport {
    $^H{complex} = 0;
}
```

```
sub in_effect {
    my $hints = (caller(1))[10];
    return $hints->{complex};
}

1;
```

Ниже приводится фрагмент программы, создающей комплексные числа:

```
use utf8;
use v5.10;
use complex;

say "1. √-25 is " => sqrt(-25);
say "2. √36 is " => sqrt( 36);

eval {
    no complex;

    say "3. √-25 is " => sqrt(-25);
    say "4. √36 is " => sqrt( 36);
} or say "Error: $@";
```

Директива no complex сбрасывает флаг $^H{complex}, запрещая извлечение квадратного корня в оставшейся области видимости. Хеш %^H имеет лексическую область видимости, поэтому после выхода из блока будет восстановлено прежнее значение. Внутри блока eval директива no complex выключает специальную обработку, поэтому sqrt(-25) вызовет ошибку:

```
1. √-25 is 5i
2. √36 is 6
Error: Невозможно выполнить sqrt для -25 at sqrt.pl line 10
```

Это упрощенный пример, модуль Math::Complex лучше справится с извлечением корней из отрицательных чисел, даже если использовать его непосредственно, а не прятать за прагмой.

Глоссарий

Если мы выделяем здесь курсивом слово или фразу, это обычно означает, что можно найти их определение в этом глоссарии. Рассматривайте это как гиперссылку.

ARGV

Имя массива, содержащего *вектор аргументов* из командной строки. Когда мы применяем пустой оператор <>, `ARGV` служит именем *дескриптора файла*, используемого при обходе аргументов, и одновременно *скаляра*, содержащего имя текущего входного файла.

ASCII

American Standard Code for Information Interchange – американская стандартная кодировочная таблица (набор семиразрядных символов, пригодный только для ограниченного представления английского текста). Часто произвольно используется для описания младших 128 значений различных наборов символов ISO-8859-X – горстки взаимно-несовместимых восьмиразрядных кодов, которые лучше всего описать как полу-ASCII. См. также *Юникод*.

AV

Сокращение от «array value» – «значение типа массив», относящееся к одному из внутренних типов данных Perl, который содержит *массив*. Тип `AV` является подклассом *SV*.

awk

Термин, используемый при редактировании, – от «awkward» (неуклюжий). По совпадению также обозначает почтенный язык обработки текста, из которого Perl позаимствовал некоторые идеи высокого уровня.

BLOCK

Синтаксическая конструкция из последовательности *инструкций* Perl, заключенной в фигурные скобки. Команды `if` и `while`, например, определяются в терминах блоков. Иногда мы также называем «блоком» лексическую область видимости, т.е. последовательность инструкций, действующих как блок, например внутри `eval` или файла, даже если команды не заключены в фигурные скобки.

BSD

Психотропный наркотик, популярный в 80-е годы, разработанный, вероятно, в университете Беркли или его окрестностях. Во многом аналогичен лекарству «System V», доступному только по рецепту, но несравнимо более полезен. (Или, как минимум, сильнее веселит.) Полное химическое название: «Berkeley Standard Distribution».

C

Язык, который многие любят за его вывернутые наизнанку определения *типов*, непостижимые правила *приоритетов* и значительную *перегрузку* механизма вызова функций. (На самом деле, изначально программисты начали переходить на C, обнаружив, что идентификаторы в нижнем регистре читаются легче, чем в верхнем.) Perl написан на C, поэтому не удивительно, что Perl позаимствовал из C некоторые идеи.

CODE

Слово, возвращаемое функцией ref в случае применения ее к ссылке на подпрограмму. См. также *CV*.

CPAN

Comprehensive Perl Archive Network – архив Perl (подробности читайте в предисловии, а также в главе 19).

CV

Внутреннее определение типа «значение кода», содержащее подпрограмму. Тип CV является подклассом *SV*.

DBM

Обозначает программы «Data Base Management» – управления базами данных, эмулирующие в совокупности *ассоциативный массив* с помощью дисковых файлов. Программы используют схему динамического хеширования для нахождения любой записи за два обращения к диску. Файлы DBM позволяют программе Perl сохранять постоянный *хеш* между запусками. Вызовом tie можно связывать переменные типа хеш с различными реализациями DBM.

dweomer

Колдовство, иллюзия, фантом, фокусничество. Употребляется, когда в Perl магические эффекты, которые производит *dwimmer*, отличаются от тех, на которые вы рассчитывали, и кажутся результатом таинственного колдовства и вмешательства нечистой силы. [От староанглийского.]

dwimmer

DWIM служит акронимом для «Do What I Mean» – делай, что я подразумеваю, принципа, согласно которому нечто должно выполнить то, чего вы хотите, не поднимая лишнего шума. Код, который действует таким образом. «Двимминг» может потребовать большого объема закулисной магии, которая (если она не остается надлежащим образом сокрытой) называется *dweomer*.

escape-последовательность – escape sequence

См. *метазнак*.

exec

Оставить программу текущего *процесса* и заменить ее другой, не завершая текущий процесс и не освобождая захваченные ресурсы (кроме прежнего образа в памяти).

feeping creaturism

Перевертыш от фразы «creeping featurism» (ползучий улучшизм)[1], означает биологическую потребность постоянно добавлять в программу новые возможности.

FIFO

First In, First Out – «первым пришел, первым ушел». См. также *LIFO*. Кроме того, прозвище для *именованных каналов*.

fileglob

Поиск *имен файлов* с помощью групповых символов, или масок («wildcard»). См. описание функции glob.

FMTEYEWTK

Far More Than Everything You Ever Wanted To Know – намного больше, чем все, что вам когда-либо требовалось узнать. Исчерпывающий трактат по какой-либо узкой теме, нечто вроде сверх-ЧаВо. Гораздо больше можно узнать у Тома.

GID

Идентификатор группы – в UNIX используется операционной системой для идентификации вас и членов вашей группы.

glob

Строго говоря – символ * интерпретатора команд, соответствующий «глобу» (группе) символов, используемый с целью сгенерировать список имен файлов. Вольно говоря – собственно применение глобов и аналогичных символов для поиска по шаблону. См. также *fileglob* и *typeglob*.

---

[1] Постоянное усложнение программы за счет мелких и ненужных улучшений. При этом не всегда улучшения ненужные! Другое дело, что они размывают изначальное видение и понимание программы. – *Прим. перев.*

grep

Происходит от старой команды редактора UNIX «Globally search for a Regular Expression and Print it» (глобальный поиск по регулярному выражению и вывод результатов). Сейчас используется в общем смысле поиска любого вида, особенно в тексте. В Perl есть встроенная функция `grep`, которая ищет в списке элементы, удовлетворяющие любому заданному критерию, тогда как программа *grep*(1) ищет в одном или нескольких файлах строки, соответствующие регулярному выражению.

GV

Внутреннее определение типа «значения glob», содержащее *typeglob*. Тип `GV` является подклассом *SV*.

HV

Сокращенно от определения типа «hash value», в котором содержится внутреннее представление хеша в Perl. Тип `HV` является подклассом *SV*.

I/O

Ввод/вывод, связанный с файлом или устройством.

IO

Внутренний объект ввода/вывода. Может также обозначать *косвенный объект* (indirect object).

IP

Internet Protocol, или Intellectual Property (интеллектуальная собственность).

IPA

India Pale Ale (индийский светлый эль). А также International Phonetic Alphabet (международный фонетический алфавит, МФА), стандартный алфавит, используемый во всем мире для записи транскрипции. Много заимствует из Юникода, включая многие комбинационные символы.

IPC

Межпроцессное взаимодействие (Interprocess Communication).

IV

Число четыре, не путать с six, любимым редактором Тома. IV означает также Integer Value (целочисленное значение) – внутреннее целочисленное значение типа, который может содержаться в *скаляре*, не путать с *NV*.

JAPH

«Just Another Perl Hacker» («еще один хакер Perl»). Заумный, но таинственный фрагмент кода Perl, который в результате выполнения дает эту строку. Часто используется для иллюстрации некоторой возможности Perl и представляет собой что-то вроде постоянного конкурса на самый непонятный код на Perl в конференциях Usenix.

LIFO

Last In, First Out – «последним пришел, первым ушел». См. также *FIFO*. LIFO обычно называется стеком.

lvaluable

Способный служить *левосторонним значением*.

Makefile

Файл, управляющий компиляцией программы. Программам на Perl обычно не требуется *Makefile* – у компилятора Perl имеется масса средств самоуправления.

man

Программа UNIX, которая выводит электронную документацию (страницы руководства).

minicpan

Зеркало *CPAN*, содержащее только последние версии дистрибутивов. Возможно, созданное с помощью `CPAN::Mini`. См. главу 19.

monger

Краткое обозначение члена группы пользователей Perl (*Perl Monger*).

mro

См. *порядок разрешения методов*.

NaN

Not a number (не число). Значение, используемое в Perl для представления некоторых недопустимых или невыразимых результатов операций с плавающей запятой.

NFS

Network File System – сетевая файловая система, позволяющая монтировать удаленную файловую систему, как если бы она была локальной.

NV

Сокращение от «Невада», ни одна часть которой никогда не будет спутана с цивилизацией. NV означает также Numeric Value (числовое значение) – внутреннее представление числа с плавающей запятой типа, который может содержаться в скаляре; не путать с *IV*.

pad

Сокращение от *scratchpad* – временная память.

PATH

Список *каталогов*, где система ищет программу, которую нужно *выполнить*. Этот список хранится в одной из *переменных среды*, которая доступна в Perl под именем `$ENV{PATH}`.

PAUSE

Perl Authors Upload SErver (*http://pause.perl.org*) – ворота для модулей в *CPAN*.

Perl mongers

Группа пользователей Perl, получившая свое название по наследству от «New York Perl mongers» – первой группы пользователей Perl. Поищите ближайшую к себе группу на сайте *http://www.pm.org*.

Pern

Именно это получится, если выполнить `Perl++` дважды. Выполнив один раз, вы завьете себе волосы. Вам придется выполнить это восемь раз, чтобы вымыть волосы. Намылить, смыть, повторить[1].

pod

Разметка, используемая для встраивания документации в код Perl. Название Pod происходит от «Plain old documentation» (простая старая документация). См. главу 23.

POSIX

Спецификация переносимого интерфейса операционных систем (Portable Operating System Interface).

pp

Внутреннее сокращение для кода «push-pop», т.е. кода на C, реализующего механизм стеков Perl.

pumpking

Обладатель *тыквы* – лицо, ответственное за работу насоса, или, по крайней мере, за его запуск. Время от времени играет роль Большой Тыквы.

PV

«pointer value», на внутреннем наречии Perl означает `char*`.

regex

См. *регулярное выражение*.

RFC

Request For Comment (просьба прокомментировать) – несмотря на звучащую в названии робость, является названием ряда важных стандартизирующих документов.

root

Суперпользователь (`UID == 0`). Кроме того, каталог файловой системы самого верхнего уровня («корень»).

RTFM

Это произносят, когда хотят, чтобы вы прочли руководство – Read The Fine Manual.

RV

RV означает внутреннее ссылочное значение (Reference Value) типа, который может содержаться в скаляре. Если вы еще не запутались, см. *IV* и *NV*.

---

[1] Здесь обыгрываются слова Pern (осоед), Perm (перманент, завивка), Pert (название шампуня). – *Прим. перев.*

script kiddie

Взломщик, который не *хакер*, его знаний хватает лишь для запуска сценариев. Не разбирающийся в сути (*cargo-cult*) программист.

sed

Уважаемый редактор Stream EDitor, из которого Perl позаимствовал некоторые свои идеи.

setgid

То же, что *setuid*, только относящееся к отдаче прав *группы*.

setuid

Обозначает программу, работающую с правами своего *владельца*, а не того, кто ее запустил (как обычно бывает). Также обозначает один из флагов режима (*битов разрешений*), который управляет этой возможностью. Этот бит должен быть явно установлен владельцем, чтобы включить данную функцию, а программа должна быть аккуратно написана, чтобы не дать больше прав, чем это необходимо.

shebang

В культуре Perl слово-гибрид, образованное из «sharp» (диез) и «bang» (восклицательный знак), обозначающее последовательность `#!`, которая сообщает системе, где найти интерпретатор команд.

slurp

Считать весь файл в строку за одну операцию.

STDERR

См. *стандартное устройство вывода ошибок*.

STDIN

См. *стандартное устройство ввода*.

STDIO

См. *стандартный ввод/вывод*.

STDOUT

См. *стандартное устройство вывода*.

struct

Ключевое слово, вводящее определение или имя структуры.

SV

Сокращение для «scalar value» – скалярное значение. Но в интерпретаторе Perl любой *объект ссылки* рассматривается как член класса, производного от SV, в объектно-ориентированном порядке вещей. Каждое значение внутри Perl передается как указатель `SV*` на языке C. *struct* SV знает свой собственный «тип объекта ссылки», а код достаточно сообразителен (надеемся), чтобы не пытаться вызывать функцию *хеша* с *подпрограммой*.

TCP

Сокращение от Transmission Control Protocol – протокол управления передачей данных. Протокол, служащий оболочкой для Internet Protocol, которая заставляет ненадежный механизм передачи пакетов выглядеть для прикладной программы как надежный *поток* байтов. (Обычно.)

TMTOWTDI

There's More Than One Way To Do It[1] – есть более одного способа сделать это – девиз Perl. Идея о том, что может найтись более одного допустимого способа решения рассматриваемой задачи программирования. (Это не значит, что «больше» всегда «лучше» или что всевозможные пути одинаково желательны, – просто не должно быть Единственно Правильного Пути.)

troff

Почтенный наборный язык, из которого Perl взял название своей переменной `$%` и который тайно используется при наборе книг серии Camel.

typedef

Определение типа в языке C.

typeglob

Использование одного идентификатора с префиксом `*`. Например, `*name` замещает любое имя из списка `$name`, `@name`, `%name`,

[1] Или «Все верблюды идут в Рим». – *Прим. ред.*

&name, или просто name, или все перечисленные имена одновременно. Как он будет интерпретироваться – как одно имя или как все имена, – зависит от контекста его применения. См. раздел «Таблицы символов и дескрипторы файлов» в главе 2.

typemap

Описание преобразований типов на языке C в типы на языке Perl и обратно в *модуле расширения*, написанном на *XS*.

UDP

User Datagram Protocol, типичный способ отправки *датаграмм* через Интернет.

UID

Идентификатор пользователя. Часто используется в контексте владения *файлом* или *процессом*.

umask

Маска для тех *битов разрешений*, которые должны быть сброшены при создании файлов и каталогов, чтобы установить политику для тех, кому мы обычно отказываем в доступе. См. функцию umask.

UNIX

Очень большой и постоянно развивающийся язык, имеющий несколько вариантов во многом несовместимого синтаксиса. На этом языке любой может выразить все тем способом, каким пожелает, что обычно и делает. Владеющие этим языком считают, что его легко освоить, поскольку его так легко приспособить к собственным целям, но различия в диалектах делают межплеменное общение почти невозможным, и путешественники часто ограничиваются подмножеством языка типа пиджин-инглиш. Чтобы быть понятым всюду, автор сценариев интерпретатора команд UNIX должен посвятить годы обретению мастерства. Многие бегут от этой кары и общаются на языке, который подобен эсперанто и называется Perl.

В древние времена UNIX использовался также для обозначения некоторого кода, который пара человек из Bell Labs написали, чтобы найти применение компьютеру PDP-7, который тогда ничем особенным не занимался.

v-строка – v-string

Строка «версии», или «вектор», задаваемая как v с последующими десятичными целыми в точечной нотации, например v1.20.300.4000. Каждое число преобразуется в *символ* с заданным порядковым значением (если чисел не менее трех, символ v может отсутствовать).

WYSIWYG

What You See Is What You Get – что видишь, то и получишь. Обычно употребляется, когда нечто, представленное на экране, соответствует тому, как оно будет выглядеть в конечном итоге, например объявления format в Perl. Также обозначает противоположность волшебству, если все работает точно так, как выглядит, например в open с тремя аргументами.

XS

Невероятно экспортируемый, чудо какой великолепный, необычайно выразительный новый язык расширения для создания внешних подпрограмм (eXternal Subroutine), написанных на C или C++.

XSUB

Внешняя *подпрограмма*, определенная на языке *XS*.

yacc

Yet Another Compiler Compiler – еще один компилятор компиляторов. Генератор анализаторов, без которого существование Perl было бы, вероятно, невозможно. См. файл *perly.y* в дистрибутиве исходного кода Perl.

автозагрузка – autoload

Загрузка по мере необходимости. (Также называется «отложенной» или «ленивой» загрузкой.) Конкретно: вызов подпрограммы AUTOLOAD в интересах подпрограммы, которая не определена.

автоинкрементирование – autoincrement

Автоматическое прибавление единицы к существующему значению; отсюда ведет свое название оператор ++. Если же единица автоматически вычитается, это называется «автодекрементированием».

автоматическая генерация – autogeneration

Возможность *перегрузки операторов* объектов, при которой поведение некоторых операторов можно предсказать, основываясь на более фундаментальных операторах. При этом предполагается, что перегруженные операторы часто имеют такие же связи между собой, как обычные операторы. См. главу 13.

автоматическое расщепление – autosplit

Автоматическое расщепление строки, как его осуществляет *ключ* (*switch*) -a при запуске с -p или -n для эмуляции *awk*. (См. также модуль AutoSplit, который никак не связан с ключом -a, но весьма тесно связан с автозагрузкой.)

алгоритм – algorithm

Четко определенная последовательность шагов, описанная достаточно ясно, чтобы их мог выполнить даже компьютер.

алфавитный – alphabetic

Тип *символов*, из которых мы составляем *слова*. В *Юникоде* это все буквы, включая все идеографические знаки и некоторые диакритические знаки, буквы-цифры, такие как римские цифры, и различные комбинационные знаки.

альтернативы – alternatives

Список возможных вариантов, из которых можно выбрать только один, как в этом вопросе: «В какую дверь вы хотите войти – A, B или C?» Альтернативы в регулярных выражениях разделяются одиночной вертикальной чертой: |. Альтернативы в обычных выражениях Perl разделяются двойной вертикальной чертой: ||. Логические альтернативы в *булевых* выражениях разделяются оператором || или or.

анонимный – anonymous

Используется для описания *объекта ссылки*, который не доступен непосредственно через именованную *переменную*. Такой объект ссылки должен быть косвенно доступен по крайней мере через одну *жесткую ссылку*. При исчезновении последней жесткой ссылки анонимный объект безжалостно уничтожается.

аргумент – argument

Некоторые данные, передаваемые *программе*, *подпрограмме*, *функции* или *методу*, и указывающие, что следует сделать. Аргумент также называют «параметром».

аргументы командной строки – command-line arguments

Значения, передаваемые вместе с именем программы, когда необходимо указать *интерпретатору команд*, что он должен выполнить *команду*. Эти значения передаются программе Perl через @ARGV.

арифметический оператор – arithmetical operator

Символ, такой как + или /, указывающий Perl на необходимость выполнения арифметических действий, которые вы должны были изучить в начальной школе.

Артистическая Лицензия – Artistic License[1]

Открытая лицензия, созданная Ларри Уоллом (Larry Wall) для Perl, чтобы максимально повысить практическую пользу Perl, его доступность и изменчивость. Текущей является версия 2.0 (*http://www.opensource.org/licenses/artistic-license.php*).

архитектура – architecture

Тип компьютера, на котором вы работаете, где «тип» означает, что все такие компьютеры используют совместимый машинный язык. Поскольку программы на Perl (обычно) являются простыми текстовыми файлами, а не исполняемыми образами, программа на Perl значительно менее чувствительна к архитектуре платформы исполнения, чем другие языки (например, C), которые *компилируются* в машинный код. См. также *платформа* и *операционная система*.

---

1 Словосочетание «artistic license» также переводится как «поэтическая вольность». – *Прим. ред.*

асинхронный – asynchronous

Касается событий или действий, относительный порядок которых во времени не детерминирован, так как слишком многое происходит одновременно. Поэтому асинхронным называют событие, о котором неизвестно, когда его следует ожидать.

ассоциативность – associativity

Определяет, какой *оператор* следует выполнить сначала – левый или правый, если дано выражение «А *оператор* В *оператор* С» и оба оператора имеют одинаковый приоритет. Операторы типа + являются ассоциативными слева, а операторы типа ** – ассоциативными справа. См. в главе 3 список операторов с указанием их ассоциативности.

ассоциативный массив – associative array

См. *хеш*. Термин «ассоциативный массив» использовался для обозначения *хешей* в Perl 4. В некоторых языках хеши называются словарями.

атом – atom

Компонент *регулярного выражения*, который может соответствовать *подстроке*, содержащей один или более символов, и рассматриваемый как неделимая синтаксическая единица любыми последующими *квантификаторами*. (В противоположность *утверждению*, соответствующему чему-то, что имеет *нулевую ширину* и не может быть квантифицировано.)

атомарная операция – atomic operation

Когда Демокрит назвал словом «атом» неделимые частицы материи, он буквально имел в виду нечто, что нельзя разрезать: *ἀ-* (не) + *-τομος* (делимое). Атомарная операция – это действие, которое нельзя прервать, а не действие, запрещенное в безъядерной зоне.

атомарный квантификатор – possessive

Так называются квантификаторы и группы в шаблонах, которые не отдают то, что им удалось заполучить. Более простой и понятный термин, чем «не допускающий возврата».

атрибут – attribute

Новая возможность, позволяющая объявлять *переменные* и *подпрограммы* с модификаторами, например `sub foo : locked method`. Кроме того, используется как другое название для *переменной экземпляра объекта*.

базовый класс – base class

Обобщенный тип *объекта*; означает класс, от которого путем *наследования* генетически производятся другие, более специализированные классы. Уважающие своих предков используют так же обозначение «надкласс» или «суперкласс».

байт – byte

Фрагмент данных из восьми *битов*.

байт-код – bytecode

Гибридный язык, на котором разговаривают андроиды, когда не хотят раскрывать свою ориентацию (см. *порядок следования байтов*). Получил название от аналогичных языков, на которых (по тем же причинам) общались между собой интерпретаторы и компиляторы в конце XX века. Эти языки отличаются тем, что представляют все как последовательность байтов, не зависящую от архитектуры.

бесплатно доступное – freely available

Означает, что за это не надо платить, но авторские права на это могут принадлежать кому-нибудь другому (например, Ларри).

бесплатно распространяемое – freely redistributable

Означает, что у вас не будет неприятностей с законом, если вы сделаете копию и дадите своим друзьям, а мы об этом узнаем. На самом деле мы бы хотели, чтобы вы раздали копии всем своим друзьям.

бесплатное программное обеспечение – freeware

Традиционно – любое программное обеспечение, которое отдают даром, особенно если при этом делают доступным также исходный код. Теперь это часто называют *программным обеспечением с открытым исходным кодом* (*open source software*). Одно время была тенденция использовать

термин *freeware* в противовес термину *open source software*, подчеркивая бесплатность программного обеспечения, выпускаемого под универсальной общественной лицензией Фонда свободно распространяемого программного обеспечения (Free Software Foundation General Public License – GPL), но это трудно оправдать этимологически.

библиотека – library

Обычно – собрание процедур. В прежние времена термин обозначал собрание процедур в файле *.pl*. В наше время чаще относится ко всей совокупности *модулей* Perl в системе.

бинарный – binary

См. *двоичный*.

бинарный оператор – binary operator

Оператор, принимающий два *операнда*.

бит – bit

Целое число в диапазоне от 0 до 1 включительно. Наименьшая возможная единица хранения информации. Одна восьмая часть *байта* или доллара. (Название старого испанского доллара «piece of eight», т.е. «из восьми частей», происходит от того, что можно было разделить его на восемь частей, каждая из которых по-прежнему считалась деньгами. Поэтому для 25-центовой монеты и сегодня сохранилось название «two bits».)

бит выполнения – execute bit

Специальная метка, сообщающая операционной системе, что ту или иную программу можно выполнить. На самом деле, в UNIX есть три бита выполнения, а который из них используется, зависит от того, владеете ли вы файлом единолично, коллективно или не владеете вообще.

битовая строка – bit string

(Битовый вектор.) Последовательность *битов*, которая в данном случае действительно рассматривается как последовательность битов.

биты разрешений – permission bits

Биты, которые устанавливает или сбрасывает *владелец* файла, чтобы разрешить или запретить доступ к нему других пользователей. Эти биты флагов являются частью *режима*, возвращаемого встроенной функцией stat по запросу информации о файле. В системах UNIX можно получить дополнительные сведения на странице руководства *ls*(1).

блок, блокирование – block

То, что делает *процесс*, когда ему приходится чего-то ждать: «Мой процесс был заблокирован в ожидании диска». Выполняя функцию существительного из другой оперы, означает фрагмент данных, имеющий размер, выбранный *операционной системой* (обычно степень двух, например 512 или 8192). Как правило, речь о данных, поступивших из дискового файла или записываемых в него.

блок, в хешах – bucket

Область *хеш-таблицы*, состоящая (потенциально) из многих записей, ключи которых «хешируются» в одно и то же значение в результате применения хеш-функции. (Это внутренняя политика, которая не должна вас беспокоить, если только вас не интересуют внутренности или политика.)

блочная буферизация – block buffering

Способ повышения эффективности ввода/вывода путем передачи сразу целого *блока*. По умолчанию Perl осуществляет поблочную буферизацию дисковых файлов. См. *буфер* и *буферизация команд*.

булев контекст – Boolean context

Особого рода *скалярный контекст*, используемый в условных операторах для определения истинности (или ложности) скалярного значения, возвращаемого выражением. Не вычисляется как строка или число. См. *контекст*.

булево значение – Boolean

Значение, которое является истиной (true) или ложью (false).

буфер – buffer

Область для временного хранения данных. *Блочная буферизация* означает, что данные передаются в место своего назначения, когда буфер оказывается заполненным. *Строковая буферизация* означает, что они

передаются, когда получена законченная строка. *Буферизация команд* означает, что данные передаются при выполнении команды `print` (или эквивалентной ей). Если вывод не буферизован, система обрабатывает байты по одному, не используя промежуточную область хранения. Это может оказаться весьма неэффективным.

буферизация команд – command buffering

Механизм, позволяющий накапливать вывод каждой *команды* Perl, а затем выталкивать все сразу единственным обращением к *операционной системе*. Он включается путем установки переменной `$|` (`$AUTOFLUSH`) в истинное значение. Используется, когда нежелательно, чтобы данные задерживались и не отправлялись туда, куда положено, что может происходить, так как по умолчанию для *файлов* и *каналов* используется *блочная буферизация*.

буферизация строковая – line buffering

Используется выходным потоком *стандартного ввода/вывода*, который выталкивает свой *буфер* после каждого *символа перевода строки*. Многие библиотеки стандартного ввода/вывода автоматически устанавливают строковую буферизацию при выводе в терминал.

вариадический – variadic

Относится к *функции*, способной принимать неопределенное число фактических аргументов.

вектор – vector

Математический жаргонный термин, обозначающий список скалярных значений.

верхний регистр – uppercase

В Юникоде к верхнему регистру относятся не только символы из категории Uppercase Letter, но и любые символы со свойством Uppercase, включая Letter Numbers и Symbols. Не путайте с *заглавным регистром*.

ветвление – fork

Порождает *процесс*, идентичный родительскому на момент зачатия, но пока не получивший собственные мысли. Поток (thread) с защищенной памятью.

взломщик – cracker

Некто, взламывающий систему безопасности в компьютерных системах. Взломщик может быть настоящим *хакером* или просто *script kiddie*.

виртуальный – virtual

Создающий видимость чего-либо, что не существует в реальности, как, например, это делает виртуальная память, не являющаяся памятью в действительности. (См. также *память*.) Противоположностью «виртуального» является «прозрачное» (transparent), что означает реальное существование чего-либо без внешнего проявления; например, прозрачная обработка Perl строк переменной длины в кодировке символов UTF-8.

висячая команда – dangling statement

Отдельная голая *команда*, без каких-либо скобок, «свисающая» с условного оператора `if` или `while`. Язык C разрешает их использование. Perl – нет.

владелец – owner

Единственный пользователь (помимо суперпользователя), имеющий абсолютный контроль над *файлом*. У файла может существовать *группа* пользователей, совместно владеющих файлом, если это разрешено фактическим владельцем. См. *биты разрешений*.

внедренные документы – here documents

Названы так по аналогии с эквивалентной конструкцией в *интерпретаторах команд*, которая делает вид, что *строки*, следующие за командой, являются отдельным *файлом*, который нужно подать на вход команды, вплоть до некоторой завершающей строки. Однако в Perl это просто причудливый способ заключить строку в кавычки.

возвращаемое значение – return value

Значение, которое создает *подпрограмма* или *выражение* при вычислении. В Perl возвращаемое значение может быть *списком* или *скаляром*.

волшебное инкрементирование — magical increment

Оператор *инкрементирования*, который умеет наращивать буквы так же, как и цифры.

волшебные переменные — magical variables

Специальные переменные с побочными эффектами, проявляющимися при обращении к ним или присваивании. Например, в Perl изменение элементов массива %ENV изменяет соответствующие переменные среды, которые будут использоваться подпроцессами. Чтение переменной $! дает номер текущей системной ошибки или сообщение для нее.

волшебство, магия — magic

Технически говоря, любая дополнительная семантика, прикрепленная к переменным, таким как $!, $0, %ENV или %SIG, либо к любой связанной переменной. Волшебство происходит, когда вы модифицируете (не слишком аккуратно) эти переменные.

восьмеричный — octal

Число по основанию 8. Допустимы только цифры от 0 до 7. Восьмеричные константы начинаются в Perl нулем, как, например в случае 013. См. также функцию oct.

временная память — scratchpad

Область, в которой конкретный вызов конкретного файла или подпрограммы хранит некоторые свои временные значения, включая переменные с лексической областью видимости.

встраивание — embedding

Когда одно содержится в другом, особенно если это необычно: «Я встроил полный интерпретатор Perl в свой редактор!»

встроенная функция — built-in

Функция, которая предопределена в языке. Даже если она скрыта в результате *переназначения*, вы всегда можете вызвать встроенную функции, если *квалифицируете* имя такой функции псевдопакетом CORE::.

вызов — invocation

Действие, направленное на вызов божества, демона, программы, метода, подпрограммы или функции, имеющее целью заставить их делать то, что, по вашему мнению, они должны делать. Обычно «call» относится к подпрограммам, а «invoke» — к методам, поскольку это звучит более модно.

вызов по значению — call by value

Механизм передачи *аргументов*, при котором *формальные аргументы* ссылаются на копии *фактических аргументов*, и *подпрограмма* не может изменить фактические аргументы, изменяя формальные. См. также *вызов по ссылке*.

вызов по ссылке — call by reference

Механизм передачи *аргументов*, при котором *формальные аргументы* непосредственно ссылаются на *фактические аргументы*, и *подпрограмма* может изменять фактические аргументы, изменяя формальные. Это означает, что формальный аргумент представляет собой *псевдоним* фактического аргумента. См. также *вызов по значению*.

вызывающий — invocant

Агент, от имени которого вызывается *метод*. Для метода *класса* вызывающим (инвокантом) является имя пакета. Для метода *экземпляра* вызывающей является ссылка на объект.

выполнить — execute

Запустить текущую *программу* или *подпрограмму*. (Не имеет отношения к встроенной функции kill, если только вы не пытаетесь запустить *обработчик сигнала*.)

выражение — expression

Все, что допускается сказать там, где требуется значение. Обычно составляется из *литералов*, *переменных*, *операторов*, *функций* и вызовов *подпрограмм*, не обязательно в указанном порядке.

высокомерие — hubris

Чрезмерная гордость — из разряда вещей, за которые Зевс мечет в нас свои молнии. Также качество, заставляющее нас писать (и сопровождать) такие программы, о кото-

рых другие не смогут сказать ничего худого. Посему является третьей великой добродетелью программиста. См. также *лень* и *нетерпеливость*.

генератор кода – code generator

Система, которая пишет для вас код на языке низкого уровня, например, код, реализующий сервер компилятора. См. *генератор программ*.

генератор программ – program generator

Система, алгоритмически создающая для пользователя код на языке высокого уровня. См. также *генератор кода*.

главный хранитель – primary maintainer

Автор, зарегистрированный на сервере *PAUSE*, обладающий правом выдавать привилегии *со-хранителя пространства имен*. Главный хранитель может передать свои привилегии другому автору, зарегистрированному на PAUSE. См. главу 19.

глобальное разрушение – global destruction

Уборка мусора для глобальных элементов (и выполнение деструкторов всех вовлеченных объектов), которая происходит по завершении работы *интерпретатора* Perl. Глобальное разрушение не следует путать с Апокалипсисом, исключая тот случай, когда они совпадают.

глобальный – global

Нечто, что видно отовсюду; обычно используется в отношении *переменных* и *подпрограмм*, которые видны в любом месте программы. В Perl подлинно глобальные переменные можно сосчитать на пальцах – большинство переменных (и все подпрограммы) существует только в текущем *пакете*. Глобальные переменные могут объявляться с помощью `our`. См. раздел «Глобальные объявления» главы 4.

голое (простое) слово – bareword

Слово достаточно неоднозначное, чтобы рассматриваться как незаконное, если действует директива `use strict 'subs'`. В отсутствие такого ограничения голое слово рассматривается, как если бы оно было заключено в кавычки.

графема – grapheme

Графен – двумерная аллотропная модификация углерода, образованная слоем атомов углерода толщиной в один атом, соединенных в гексагональную двумерную кристаллическую решетку. *Графема*, более полное название *«кластер графемы»* – единый *символ*, видимый пользователем, который в свою очередь может состоять из нескольких символов (*кодов*). Например, возврат каретки и перевод строки образуют единственную графему, состоящую из двух символов, а «ȫ» – единственная графема, но состоящая из одного, двух или даже трех символов, в зависимости от нормализации.

группа – group

Множество пользователей, которому вы принадлежите. В некоторых операционных системах (например, UNIX) вы можете давать определенные права доступа к файлам остальным членам своей группы.

дамп памяти – core dump

Тело *процесса* в виде файла, сохраненного в *рабочем каталоге* процесса, обычно как результат некоторых видов фатальных ошибок.

данные экземпляра – instance data

См. *переменная экземпляра*.

датаграмма – datagram

Пакет данных, например сообщение *UDP*, которое (с точки зрения участвующих программ) может независимо пересылаться по сети. (На самом деле, все пакеты посылаются независимо на уровне *IP*, но *поточные* протоколы вроде *TCP* скрывают это от вашей программы.)

двоичный – binary

Относящийся к числам по основанию 2. Это означает, что, по существу, есть два числа, 0 и 1. Употребляется также для описания «нетекстового» файла, возможно, потому, что такой файл использует все двоичные разряды своих байтов. С приходом *Юникода* такое различие, уже изна-

чально подозрительное, еще более утрачивает свое значение.

двойная жизнь – dual-lived

Некоторые модули распространяются одновременно и в составе *стандартной библиотеки*, и через *CPAN*. Развитие этих модулей может идти двумя путями, так как дорабатываться может любая из двух версий. В настоящее время стоит вопрос о том, как упорядочить подобные ситуации.

декрементирование – decrement

Вычитание значения из переменной, например, «декрементировать $x» (что означает вычесть 1 из значения) или «декрементировать $x на 3».

дерево синтаксического анализа – parse tree

См. *синтаксическое дерево.*

дескриптор каталога – directory handle

Имя, представляющее конкретный открытый для чтения каталог – вплоть до его закрытия. См. функцию opendir.

дескриптор файла – filehandle

Идентификатор (не обязательно связанный с действительным именем файла), представляющий конкретный открытый файл до тех пор, пока не будет закрыт. Если предполагается открывать и закрывать несколько разных файлов подряд, удобно открывать их все через один и тот же дескриптор файла, а не писать отдельный код для обработки каждого файла.

деструктор – destructor

Особый *метод*, вызываемый, когда объект собирается *разрушить* себя. Метод DESTROY ничего фактически не разрушает; Perl просто *запускает* этот метод, когда *класс* нуждается в какой-нибудь чистке, связанной с разрушением.

динамическая область видимости – dynamic scoping

Динамическая область видимости делает переменные видимыми во всей оставшейся части блока, в котором они впервые использованы, и в любых *подпрограммах*, вызываемых оставшейся частью *блока*. Значения переменных с динамической областью видимости могут временно изменяться (и неявно восстанавливаться позже) с помощью оператора local. (Сравните с *лексической областью видимости.*) Более вольная трактовка обозначает, что подпрограмма в процессе вызова другой подпрограммы «содержит» эту последнюю на этапе *выполнения.*

директива – directive

Директива *pod*. См. главу 23.

дистрибутив – distribution

Стандартный укомплектованный выпуск системы программного обеспечения. Обычное использование предполагает наличие исходного текста. Если это не так, дистрибутив называется двоичным (binary-only).

дисциплина – discipline

Некоторым она необходима, а некоторые ее избегают. В Perl – это прежнее название фильтров ввода/вывода (I/O layer).

домашний каталог – home directory

Каталог, в который система переносит пользователя после регистрации. В системах UNIX его имя часто помещается в $ENV{HOME} или $ENV{LOGDIR} при *регистрации*, но можно найти его и посредством (getpwuid($<))[7]. (Некоторые платформы не имеют понятия домашнего каталога.)

доступа, методы – accessor methods

Методы, предназначеные для косвенного просмотра или обновления состояния объекта (его переменных экземпляра).

единица компиляции – compilation unit

Файл (или *строка* в случае eval), *компилируемый* в данный момент.

жадный – greedy

Подшаблон, квантификатор *которого* хочет найти как можно больше.

жесткая ссылка – hard reference

Скалярная величина, содержащая фактический адрес *объекта ссылки*, которая учитывается *счетчиком ссылок* объекта ссылки. (Некоторые жесткие ссылки хранятся внутренне, например неявная ссылка в одной из позиций переменной *typeglob* на соответствующий объект ссылки.) Же-

стская ссылка отличается от *символической ссылки*.

заглавный регистр – titlecase

Регистр прописных символов, за которыми следуют строчные символы, а не символы верхнего регистра. Иногда называют регистром предложения. В английском алфавите не используется заглавный регистр Юникода, но правила оформления регистра символов в названиях на английском языке гораздо сложнее, чем простое приведение к верхнему регистру первых символов всех слов.

заголовочный файл – header file

Файл, содержащий некие определения, которые необходимо включить «во главе» своей программы, чтобы произвести некоторые скрытые операции. Заголовочный файл C имеет расширение *.h*. Perl, по правде говоря, не имеет заголовочных файлов, хотя традиционно иногда использовал оттранслированные файлы *.h* с расширением *.ph*. См. `require` в главе 27. (Заголовочные файлы заменены механизмом *модулей*.)

замещение – overriding

Сокрытие или отключение некоторого другого определения с тем же именем. (Не путать с *перегрузкой*, которая добавляет определения, двусмысленность которых должна быть разрешена каким-то другим способом.) Чтобы еще сильнее запутать вопрос, мы используем слово с двумя перегруженными определениями: для описания того, как можно определить собственную *подпрограмму* и скрыть встроенную *функцию* с таким же именем (см. раздел «Замещение встроенных функций» главы 11), и для описания того, как можно определить в *производном классе* замещающий *метод*, чтобы скрыть метод *базового класса* с тем же именем (см. главу 12).

замыкание – closure

Анонимная подпрограмма, которая, при создании ссылки на нее на этапе выполнения, продолжает отслеживать видимые извне *лексические переменные* даже после того, как эти лексические переменные должны были выйти из области видимости. Название «замыкание» вызвано тем, что подобное поведение анонимных функций дает математикам ощущение приятного замыкания[1].

запись – record

Набор взаимосвязанных значений в *файле* или *потоке*, часто ассоциированных с уникальным полем *ключа*. В UNIX обычно соответствует *строке* или группе строк, оканчивающейся пустой строкой (иначе говоря, «абзацу»). Каждая строка файла */etc/passwd* представляет собой запись со сведениями о пользователе, ключом которой служит имя пользователя.

зарезервированные слова – reserved words

Слова имеющие особое значение для *компилятора*, например `if` или `delete`. Во многих языках (но не в Perl) не допускается употребление зарезервированных слов для других целей. (В конце концов, потому они и зарезервированы.) В Perl нельзя использовать их только в *метках* и *дескрипторах файлов*. Также называются «ключевыми словами».

захват – capturing

Применение круглых скобок для выделения *подшаблонов* в *регулярных выражениях* с целью сохранить найденную *подстроку* как *ссылку на найденный текст* (*backreference*). (Захваченные строки возвращаются в виде списка в *списочном контексте*.) См. главу 5.

зернистость – granularity

Говоря абстрактно, размер фрагментов, с которыми вы работаете.

значение – value

Фактические данные, в отличие от переменных, ссылок, ключей, индексов, операторов и всего прочего, что нужно для доступа к значению.

---

1 Слово closure в английском языке имеет много значений, в том числе: облегчение, исцеление, катарсис. – *Прим. ред.*

значение по умолчанию — default

Значение, которое используется, если не задано никакое иное

зомби — zombie

Умерший (завершившийся) процесс, родители которого не получили надлежащего извещения о его кончине посредством вызова wait или waitpid. При выполнении ветвления fork следует прибирать за своими порожденными процессами после их завершения, иначе таблица процессов переполнится и ваш системный администратор будет вами недоволен.

идентификатор — identifier

Сформированное в соответствии с правилами имя, применяемое почти для всего, что может интересовать компьютерную программу. Многие языки (в том числе Perl) допускают идентификаторы, которые начинаются буквой и содержат буквы и цифры. Perl считает символ подчеркивания буквой. (В Perl есть также более сложные имена, например *квалифицированные* имена.)

именованный канал — named pipe

Канал, имеющий имя на уровне *файловой системы*, к которому могут иметь доступ два несвязанных *процесса*.

импорт, импортировать — import

Получить доступ к символам, экспортируемым другим модулем. См. use в главе 27.

имя команды — command name

Имя программы, выполняемой в данный момент, как оно было набрано в командной строке. В языке C *имя команды* передается как первый аргумент командной строки. В Perl оно содержится в переменной $0.

имя файла — filename

Это имя находится в *каталоге* и может использоваться функцией open, когда мы сообщаем *операционной системе*, какой именно файл собираемся открыть, чтобы связать файл с *дескриптором файла*, который и будет представлять этот файл в нашей программе, пока мы не закроем его.

индекс — subscript

Значение, указывающее на позицию конкретного *элемента в массиве*.

индексация — indexing

В прежние времена состояла в поиске *ключа* в настоящем справочнике (например, телефонной книге), но сейчас это просто применение любого ключа или позиции для поиска соответствующего *значения*, даже если никакого индекса (указателя) не существует. Дело дошло до того, что функция index в Perl просто находит позицию (индекс) одной строки в другой.

инкапсуляция — encapsulation

Завеса абстракции, отделяющая *интерфейс* от *реализации* и требующая, чтобы любой доступ к состоянию объекта осуществлялся только через методы.

инкрементирование — increment

Увеличение какого-либо значения на 1 (или другое число, если оно указано).

инструкция — statement

Инструкция, сообщающая компьютеру, что делать дальше, как инструкция в рецепте: «Добавить мармелад в тесто и перемешивать, пока не перемешается». Инструкция отличается от *объявления*, которое предписывает компьютеру не сделать что-либо, а узнать что-либо.

инструкция управления циклом — loop control statement

Любая инструкция в теле цикла, которая может досрочно завершить цикл или пропустить *итерацию*. Как правило, лучше не пробовать выполнить это на американских горках.

интерполяция — interpolation

Вставка скалярного или списочного значения в середину другого значения так, будто оно было там всегда. В Perl интерполяция переменных осуществляется в двойных кавычках и шаблонах, а интерполяция списков происходит при создании списка значений для передачи списочному оператору или другой конструкции, принимающей *список*.

интерполяция переменных — variable interpolation

Интерполяция переменной скаляра или массива в строку.

интерпретатор — interpreter

Строго говоря, это программа, которая читает вторую программу и делает то, что эта вторая программа говорит, непосредственно, не преобразуя сначала эту программу в другую форму (чем занимаются *компиляторы*). Perl не является интерпретатором согласно этому определению, поскольку содержит некий компилятор, который преобразует программу в более пригодную для выполнения форму (*синтаксические деревья*) внутри самого процесса *perl*, которую затем интерпретирует система этапа выполнения Perl.

интерпретатор команд — shell

Интерпретатор командной строки. Программа, которая выдает интерактивное приглашение, принимает одну или несколько *строчек* ввода и выполняет указанные программы, передавая каждой из них надлежащие *аргументы* и входные данные. Интерпретаторы команд способны также выполнять сценарии, состоящие из таких команд. Распространенные интерпретаторы команд UNIX: интерпретатор Борна (*/bin/sh*), интерпретатор C (*/bin/csh*), а также интерпретатор Корна (*/bin/ksh*). Строго говоря, Perl интерпретатором команд не является, поскольку не интерактивен (хотя программы, написанные на Perl, могут быть интерактивными).

интерфейс — interface

Услуги, которые некий код обещает предоставлять всегда, в отличие от *реализации*, которую он может изменить, когда того пожелает.

инфиксный оператор — infix

Оператор, который располагается между своими *операндами*, например умножение: `24 * 7`.

исключение — exception

Причудливый термин, обозначающий ошибку. См. *фатальная ошибка*.

исполняемый файл — executable file

Файл, помеченный особым образом, говорящим *операционной системе*, что его можно выполнять как программу.

истина — true

Любое скалярное значение, отличное от `0` или `""`.

итератор — iterator

Специальная программная штуковина, которая следит за тем, в каком месте того, что вы пытаетесь обойти, вы находитесь. Цикл `foreach` в Perl содержит итератор, как и хеш, что позволяет применить к нему `each`.

итерация — iteration

Многократное выполнение чего-либо.

канал — pipe

Прямое *соединение*, которое подает вывод одного *процесса* на ввод другого без применения промежуточного временного файла. Когда канал установлен, соответствующие два процесса могут осуществлять чтение и запись, как в обычный файл, с некоторыми ограничениями.

канонический — canonical

Приведенный к стандартной форме для облегчения сравнения.

каталог — directory

Особый файл, содержащий другие файлы. В некоторых *операционных системах* такие файлы могут называться «папками», «ящиками» и т.д.

квалифицированный — qualified

С указанием полного имени. Символ `$Ent::moot` квалифицирован; `$moot` не квалифицирован. Полностью квалифицированное имя задается от каталога верхнего уровня.

квантификатор — quantifier

Компонент *регулярного выражения*, указывающий число повторений предшествующего *атома*.

класс — class

Определенный пользователем тип, реализованный в Perl через *пакет*, который

предоставляет (непосредственно или в результате наследования) *методы* (т.е. *подпрограммы*) для работы с *экземплярами* класса (его объектами). См. также *наследование*.

класс символов — character class

Список символов, заключенный в квадратные скобки и указывающий в *регулярных выражениях*, что любой символ из данного набора может находиться в данном месте. Более широко – любой предопределенный набор символов, используемый таким образом.

кластер — cluster

Заключенный в круглые скобки *подшаблон*, используемый для объединения частей *регулярного выражения* в единый *атом*.

кластер ключей — switch cluster

Объединение нескольких ключей командной строки (например, -a -b -c) в один ключ (т.е. -abc). Любой ключ с дополнительным *аргументом* должен быть последним в кластере.

клиент — client

В сетевом взаимодействии – *процесс*, инициирующий контакт с процессом сервера, чтобы обменяться данными и, возможно, получить обслуживание.

ключ — key

Строковый индекс *хеша*, используемый для поиска *значения*, связанного с этим ключом.

ключ — switch

Параметр, передаваемый в командной строке с целью повлиять на ход выполнения программы, и обычно предваряемый знаком «минус». Слово может также использоваться как кличка для *переключателя*.

ключевое слово — keyword

См. *зарезервированные слова*.

код символа — codepoint

Целое число, используемое компьютером для представления данного символа. Коды ASCII-символов находятся в диапазоне от 0 до 127; коды символов Юникода находятся в диапазоне от 0 до 0x1F_FFFF; а коды символов, поддерживаемые языком Perl, – в диапазоне от 0 до 2^{32}–1 или от 0 до 2^{64}–1 в зависимости от размерности целых чисел на конкретной архитектуре. В культуре Perl их иногда называют *порядковыми значениями*.

команда — command

В языке *интерпретатора команд* означает синтаксическое сочетание имени программы с ее аргументами. В более широком смысле – любые данные, введенные в интерпретатор команд, заставившие его что-то сделать. Еще более вольно – *инструкция* Perl, которая может начинаться *меткой* и обычно заканчивается точкой с запятой.

комбинационный символ — combining character

Любой символ из главной категории Combining Mark (\p{GC=M}), который может занимать или не занимать позицию при выводе. Некоторые комбинационные символы невидимы в принципе. Последовательность комбинационных символов и основной символ графемы, за которым они следуют, вместе образуют единый, видимый пользователем, символ, который называется *графемой*. Многие, но не все, диакритические знаки являются комбинационными символами, и наоборот.

комментарий — comment

Ремарка, не оказывающая влияния на смысл программы. В Perl комментарии начинаются символом # и продолжаются до конца строки.

компилятор — compiler

Строго говоря, это программа, которая «пережевывает» другую программу и «выплевывает» файл, содержащий программу в «пригодном для выполнения» виде, обычно в виде машинных инструкций. Программа *perl*, согласно этому определению, не является компилятором, но в ней содержится некий компилятор, который принимает программу и преобразует ее в более удобную для выполнения форму (*синтаксические деревья*) внутри самого процесса *perl*, которую затем интерпретирует *интерпретатор*. Однако существуют расширяющие *модули*, которые придают

Perl больше черт поведения «настоящего» компилятора. См. главу 16.

компиляция — compile

Процесс превращения исходного кода в представление, понятное компьютеру. См. *фаза компиляции*.

конвейер — pipeline

Ряд *процессов*, соединенных каналами, где каждый процесс подает свой вывод на вводе следующего.

конец файла — EOF

Конец файла. Иногда метафорически используется как строка-терминатор *внедренного документа*.

конкатенация — concatenation

Процедура, в ходе которой нос одного кота приклеивается к хвосту другого, а также аналогичная операция для двух *строк*.

конструирование — construct

Создание *объекта* с помощью конструктора.

конструктор — constructor

Любой *метод класса*, *метод экземпляра* или *подпрограмма*, которые создают, инициализируют, освящают и возвращают объект. Иногда мы вольно используем этот термин для обозначения *формирователя*.

конструкция — construct

Синтаксический элемент, составленный из более мелких деталей.

контекст — context

Окружение или среда. Контекст, заданный окружающим кодом; определяет, данные какого типа должно, как предполагается, вернуть конкретное *выражение*. Существует три основных контекста: *списочный контекст*, *скалярный контекст* и *пустой контекст*. Скалярный контекст иногда подразделяется на *булев контекст*, *числовой контекст*, *строковый контекст* и *пустой контекст*. Есть и «безразличный» контекст (о котором, если вам интересно, говорится в главе 2).

контекст массива — array context

Устаревшее выражение, обозначающее то, что правильнее называть *списочным контекстом*.

контрольная точка, точка останова — breakpoint

Место программы, в котором отладчику приказано остановить выполнение, чтобы можно было осмотреться и выяснить, не произошло ли чего-нибудь неправильного.

контрольное выражение — watch expression

Выражение, при изменении значения которого происходит остановка в контрольной точке в отладчике Perl.

косвенность — indirection

Если нечто в программе представляет собой не значение, которое нам нужно, а указывает на местонахождение этого значения, это косвенность. Она может осуществляться с помощью *жестких ссылок* или *символических ссылок*.

косвенный дескриптор файла — indirect filehandle

Выражение, значение которого можно использовать в качестве *дескриптора файла*: *строка* (имя дескриптора файла), *typeglob*, *ссылка* на typeglob или объект *ввода/вывода* низкого уровня.

косвенный объект — indirect object

В английской грамматике так называют короткий оборот с существительным между глаголом и его прямым дополнением, указывающий на того, к кому относится действие (косвенное дополнение). В языке Perl `print STDOUT "$foo\n";` можно понять как «глагол косвенное_дополнение дополнение», где `STDOUT` является получателем действия `print`, а `"$foo"` — выводимым объектом. Аналогично при вызове *метода* можно поместить вызывающего между методом и его аргументами:

```
$gollum = new Pathetic::Creature
"Smeagol";
give $gollum "Fisssssh!";
give $gollum "Precious!";
```

кракозябры — mojibake

Когда вы говорите на одном языке, а компьютер думает, что вы говорите на другом. Из-за ошибок трансляции, когда вы передаете текст в кодировке UTF-8, например, а компьютер думает, что имеет дело с кодировкой Latin-1, он выведет эти самые кракозябры. На японском этот термин записывается как 「文字化け」 и означает «испорченные символы». Произношение в стандартном фонетическом алфавите IPA записывается как [modzibake] и читается примерно так «мо-джи-ба-ке».

культ даров небесных — cargo cult

Копирование фрагментов программного кода без его понимания, но со слепой верой в его ценность. Этот термин происходит из доиндустриальных культур, в которых людям приходится сталкиваться с артефактами, оставленными исследователями или колонизаторами из технологически развитых культур. См. «The Gods Must Be Crazy» («Боги наверное сошли с ума»).

кэш — cache

Хранилище данных. Вместо того, чтобы многократно производить дорогостоящие вычисления, чтобы получить один и тот же результат, лучше вычислить его один раз и сохранить в кэше.

левостороннее значение — lvalue

Термин, которым законники языка обозначают адрес памяти, которому можно присваивать новые *значения*, например, *переменная* или *массив*. Буква «l» означает «left», т.е. левую часть присваивания, типичного места для левосторонних значений. *lvaluable* в отношении функций или выражений означает те из них, которым могут присваиваться значения, как, например, в pos($x) = 10.

лексема — lexeme

Иное название маркера (token).

лексема — token

Морфема в языке программирования, наименьшая единица текста с семантической значимостью.

лексическая область видимости — lexical scoping

Изучение «Oxford English Dictionary» в микроскоп. (Известна также как *статическая область видимости*, поскольку словари меняются медленно.) Аналогично рассматривание переменных, хранимых в личном словаре (пространстве имен), для каждой области видимости, которые видны только от точки объявления до конца лексической области видимости, в которой они определены. Синоним – *статическая область видимости*. Антоним – *динамическая область видимости*.

лексическая переменная — lexical variable

Переменная, на которую действует *лексическая область видимости*, объявленная с помощью my или state. Часто называется просто «lexical». (Ключевое слово our объявляет переменную с лексической областью видимости для глобальной переменной, которая сама не является лексической переменной.)

лексический анализ — lexical analysis

Иное название для *разбиения на лексемы* (*tokenizing*).

лексический анализатор — lexer

Иное название для *tokener*.

лексический анализатор — tokener

Модуль, разбивающий текст программы на последовательность *лексем* для последующего анализа синтаксическим анализатором.

лень — laziness

Качество, заставляющее прикладывать большие усилия, чтобы сократить общий расход энергии. Оно заставляет писать экономящие труд программы, которые окажутся полезными другим людям, и документировать то, что вы написали, чтобы не приходилось слишком много отвечать на вопросы. Посему это первая великая добродетель программиста. Посему и эта книга. См. также *нетерпеливость* и *высокомерие*.

литерал – literal

Лексема в языке программирования, например число или *строка*, которые дают фактическое *значение*, а не просто представляют возможные значения, как это делает *переменная*.

логический оператор – logical operator

Символы, представляющие понятия «и», «или», «исключающее или» и «не».

ложь – false

В Perl это любое значение, которое выглядит как "" или "0", если вычислено в строковом контексте. Поскольку неопределенные значения вычисляются как "", все неопределенные значения ложны, но не все ложные значения являются неопределенными.

локальный – local

Не всегда означает одно и то же. Глобальная переменная Perl может быть локализована внутри *динамической области видимости* посредством оператора local.

массив – array

Упорядоченная последовательность *значений*, хранимых так, что к любому из них можно легко обратиться по *целочисленному индексу*, задающему *смещение* значения в последовательности.

метазнак – metasymbol

Нечто, что мы назвали бы *метасимволом*, если бы не то обстоятельство, что это последовательность из более чем одного символа. Обычно первым символом этой последовательности является настоящий метасимвол, который заставляет остальные символы метазнака тоже вести себя особым образом.

метасимвол – metacharacter

Символ, предполагающий необычную интерпретацию. Какие именно символы должны восприниматься как метасимволы, в значительной мере зависит от контекста. В каждом конкретном *интерпретаторе команд* имеются свои метасимволы, заключенные в двойные кавычки, *строки* в Perl имеют другие метасимволы, а шаблоны *регулярных выражений* имеют все метасимволы двойных кавычек плюс свои собственные.

метка – label

Имя, которое мы даем *инструкции*, чтобы можно было ссылаться на нее в любом месте программы.

метка цикла – loop label

Некий ключ или имя, закрепленные за циклом, чтобы инструкции управления циклом могли указывать, каким циклом они управляют.

метод – method

Определенного рода действие, которое объект может выполнить, если попросить его об этом. См. главу 12.

метод класса – class method

Метод, для которого вызов производится от имени *пакета*, а не *ссылки на объект*. Метод, ассоциируемый с *классом* в целом.

метод экземпляра – instance method

Метод объекта, как противоположность методу *класса*.

Метод, *инвокантом* которого является объект, а не имя пакета. Каждый объект класса обладает всеми методами этого класса, т.е. метод экземпляра применяется к конкретному экземпляру класса, а не ко всем экземплярам. См. также *метод класса*.

меченый – tainted

Относится к данным, полученным из грязных рук пользователя, на которые поэтому не может полагаться надежная программа. Perl осуществляет проверку меченых данных, если выполняется программа *setuid* (или *setgid*) либо если указан ключ -T.

минимализм – minimalism

Вера в то, что «малое прекрасно». Парадоксально, но если сказать что-то на маленьком языке, оно становится большим, а если сказать что-то на большом языке, оно становится маленьким. Решайте сами.

многомерный массив – multidimensional array

Массив, требующий указать несколько индексов, чтобы найти один элемент. Perl реализует их с помощью *ссылок* – см. главу 9.

множественное наследование – multiple inheritance

Черты, приобретенные от матери и отца, смешанные непредсказуемым образом. (См. *наследование* и *одиночное наследование*.) В языках программирования (включая Perl) представление о том, что у данного класса может быть несколько прямых предков, или *базовых классов*.

модификатор – modifier

См. *модификатор инструкции*, *модификатор регулярного выражения* и *модификатор левостороннего значения*, не обязательно в указанном порядке

модификатор инструкции – statement modifier

Условие или *цикл*, помещаемые после *инструкции*, а не перед ней, если вы понимаете, о чем мы говорим.

модификатор левостороннего значения – lvalue modifier

Адъективированная (играющая роль прилагательного) псевдофункция, которая искажает смысл *левостороннего значения* некоторым декларативным образом. В настоящее время есть три модификатора левосторонних значений: `my`, `our` и `local`.

модификатор регулярного выражения – regular expression modifier

Параметр для шаблона или подстановки, например `/i`, для придания шаблону нечувствительности к регистру.

модуль – module

Файл, определяющий *пакет* с (почти) тем же именем, который может либо экспортировать символы, либо функционировать как класс *объекта*. (Главный файл модуля *.pm* может также загружать другие файлы для поддержки модуля.) См. встроенную функцию `use`.

модуль – modulus

Целочисленный делитель, если нас интересует остаток, а не частное.

мягкая ссылка – soft reference

См. *символическая ссылка*.

на проходе – en passant

Когда *значение* изменяется во время копирования. [От французского «на проходе», как в необычном маневре взятия пешки в шахматах.]

надкласс (суперкласс) – superclass

См. *базовый класс*.

наследование – inheritance

То, что мы получаем от своих предков – генетически или иным способом. Если нечто представляет собой класс, то предки этого называются *базовыми классами*, а потомки – *производными классами*. См. *одиночное наследование* и *множественное наследование*.

нетерпеливость – impatience

Гнев, который вы испытываете, когда компьютер ведет себя лениво. Это заставляет писать программы, которые не просто реагируют на наши потребности, но предвосхищают их. Или по крайней мере делают вид. Посему является второй великой добродетелью программиста. См. также *лень* и *высокомерие*.

нижний регистр – lowercase

В Юникоде к нижнему регистру относятся не только символы из категории Lowercase Letter, но и любые символы со свойством Lowercase, включая символы из категорий Modifier Letters, Letter Numbers, а также некоторые символы из категорий Other Symbols и Combining Mark.

номер ошибки – errno

Номер ошибки, возвращаемый при неудаче системного вызова. Perl ссылается на ошибку через `$!` (или `$OS_ERROR`, если вы используете модуль `English`).

номер строки – line number

Число строк, прочитанных до сих пор, плюс одна. Perl ведет отдельную нумера-

цию строк для каждого открытого им исходного или входного файла. Текущий номер строки файла исходного текста представлен в `__LINE__`. Номер текущей строки ввода (для файла, который последним читался через `<FH>`) представляется переменной `$.` (`$INPUT_LINE_NUMBER`). Многие сообщения об ошибках содержат оба значения, если они доступны.

нормализация – normalization

Преобразование текстовой строки в альтернативное, но эквивалентное *каноническое* (или совместимое) представление, в котором строки можно сравнивать, чтобы определить их равенство. Юникод определяет четыре различные формы нормализации: NFD, NFC, NFKD и NFKC.

носильщик – porter

Некто, «переносящий» программное обеспечение с одной *платформы* на другую. Перенос программ, написанных на платформозависимых языках, таких как C, может быть трудным делом, но перенос таких программ, как Perl, стоит перенесенных мучений.

нулевая ширина – zero width

Утверждение в подшаблоне, соответствующее *нулевой строке* между символами.

нулевой символ – null character

Символ со значением ASCII «ноль». Используется в C для завершения строк, но в Perl строка может содержать символы `null`.

нумификация – numification

(Иногда произносится как *нуммификация*.) На жаргоне Perl обозначает неявное преобразование в число; связанный глагол – *нумифицировать*. Слово нумификация было придумано для рифмования со словом мумификация, а глагол *нумифицировать* – с глаголом *мумифицировать*. Не имеет отношения к религиозному «numen, numina, numinous». Первоначально мы допустили ошибку, опустив в слове «nummification» одну букву «m», и многие просто привыкли к нашему необычному правописанию. Видимо, когда придумали название заголовка `HTTP_REFERER` с одной пропущенной буквой, наше правописание было где-то поблизости.

обертка – wrapper

Программа или подпрограмма, которая запускает за нас другую программу или подпрограмму, частично модифицируя ее входные и выходные данные для лучшего соответствия нашим задачам.

область видимости – scope

Как далеко видна переменная. В Perl есть два механизма видимости. *Динамическая область видимости переменных* `local` означает, что оставшаяся часть *блока* и любые *подпрограммы*, вызываемые в оставшейся части блока, могут видеть переменные, локальные для блока. *Лексическая область видимости* переменных `my` означает, что оставшаяся часть блока может видеть переменную, но другие подпрограммы, вызываемые блоком, не могут видеть переменную.

обработка исключительных ситуаций – exception handling

Реакция программы на возникновение ошибки. Механизм обработки исключительных ситуаций в Perl реализуется оператором `eval`.

обработчик – handler

Подпрограмма или *метод*, вызываемые в ответ на какое-то внутреннее событие, например *сигнал*, или при вызове перегруженного *оператора*. См. также *обратный вызов*.

обработчик сигнала – signal handler

Подпрограмма, которая вместо того, чтобы удовлетвориться тем, что ее вызовут обычным образом, сидит и ждет грома среди ясного неба, чтобы соизволить *выполниться*. В Perl гром среди ясного неба носит название сигнала, а посылаются они с помощью встроенной функции `kill`. См. описание хеша `%SIG` в главе 25 и раздел «Сигналы» главы 15.

обратная совместимость – backward compatibility

Означает сохранение возможности пользоваться старыми программами, поскольку

мы не устранили функции или ошибки, на которых они основывались.

обратный вызов – callback

Обработчик, регистрируемый в какой-либо части программы в надежде, что другая часть программы *запустит* его при появлении некоторого события, представляющего интерес.

общественное достояние – public domain

Нечто, владельцем чего не является никто. Perl защищен авторским правом и *не* является общественным достоянием – он просто *бесплатно доступен* и *бесплатно распространяется*.

объект – object

Экземпляр класса. Нечто, «знающее», что собой представляет определенный пользователем тип (класс) и что оно может делать согласно этому классу. Программа может попросить объект что-то сделать, а объект будет решать, хочет он это делать или нет. Одни объекты более любезны, чем другие.

объект ссылки – referent

То, на что указывает *ссылка*, обладающее (или не обладающее) именем. Обычными объектами ссылок являются *скаляры*, *массивы*, *хеши* и *подпрограммы*.

объявление – declaration

Утверждение о том, что нечто существует; возможно, описывающее это нечто, но не содержащее никаких указаний на время, место и способ применения этого. Объявление – это как часть рецепта, где сказано: «две чашки муки, одно большое яйцо, четыре-пять головастиков...». Противоположность объявлению – *инструкция*. Обратите внимание, что некоторые объявления одновременно служат инструкциями. Объявление подпрограммы действует так же, как определение, если прилагается ее тело.

ограничитель – delimiter

Символ или *строка*, устанавливающие границы текстового объекта произвольного размера, которые не надо путать с *разделителем* (*separator*) или *терминатором* (*terminator*). «Ограничивать» в действительности значит «окружать» или «заключать» (подобно данным круглым скобкам).

одиночное наследование – single inheritance

Черты, которые некто приобрел от матери, если она сообщила ему, что у него нет отца. (См. также *наследование* и *множественное наследование*.) В языках программирования представление о неполовом размножении классов, вследствие какового у данного класса может быть только один прямой предок, или *базовый класс*. Perl не накладывает такого ограничения, хотя, программировать на Perl можно и в таком стиле, если есть желание.

однострочник – one-liner

Целая компьютерная программа, втиснутая в одну строку.

оккультный знак – sigil

Знак, используемый в магии. В Perl – символ перед именем переменной, такой как $, @ или %.

операнд – operand

Выражение, к значению которого применяется *оператор*. См. также *приоритет*.

оператор – operator

Штука, которая преобразует некоторое число входных значений в некоторое число выходных значений, часто встраиваемая в язык с помощью специального синтаксиса или символа. Этот самый оператор может иметь определенные ожидания относительно того, какие *типы* данных мы передаем ему в качестве аргументов (*операндов*) и какого типа данные собираемся получить от него.

оператор объявления – declarator

Нечто, сообщающее вашей программе, какого рода переменные вы хотели бы получить. Perl не требует объявления переменных, но вы можете использовать `my`, `our` или `state`, чтобы указать, что хотели бы получить кое-что, отличное от того, что будет создано по умолчанию.

оператор отношения — relational operator

Оператор, который сообщает, является ли *истинным* для пары операндов некоторое конкретное отношение порядка. В Perl есть операторы отношения для чисел и строк. См. *сортирующая последовательность*.

оператор прямого доступа к памяти — address operator

Некоторые языки работают непосредственно с адресами величин в памяти, но это игра с огнем. Perl предоставляет комплект асбестовых перчаток для работы с памятью. В Perl ближе всего к оператору адреса стоит оператор обратной косой черты, но он дает *жесткую ссылку*, что значительно безопаснее, чем адрес памяти.

оператор тестирования файла — file test operator

Встроенный унарный оператор, используемый для проверки *истинности* какого-либо утверждения о файле; например, `-o $filename` проверяет, являетесь ли вы владельцем файла.

операционная система — operating system

Особая программа, которая выполняется на голой машине и скрывает детали управления *процессами* и *устройствами*. Обычно употребляется в более вольном смысле, обозначая определенную культуру программирования. Вольный смысл может интерпретироваться на разных уровнях конкретности. Одной крайностью будет сказать, что все версии UNIX и UNIX-подобных систем суть одна и та же операционная система (очень многих этим расстроив, особенно законоведов и адвокатов). Другая крайность – сказать, что данная конкретная версия операционной системы данного конкретного поставщика отличается от всех других версий операционных систем любых поставщиков. Perl значительно лучше переносится между операционными системами, чем многие другие языки. См. также *архитектура* и *платформа*.

опережающая проверка — lookahead

Утверждение, которое заглядывает в строку правее текущего найденного соответствия.

определенный — defined

Имеющий значение. Perl полагает, что некоторые вещи из тех, которые люди пытаются делать, лишены смысла, в частности использование переменных, которым не присвоено *значение*, и осуществление некоторых действий над данными, которых нет. Например, при попытке чтения за пределами конца файла Perl возвращает неопределенное значение. См. также *false* и оператор `defined` в главе 27.

опции — options

См. *ключи* или *модификаторы регулярных выражений*.

освящение — bless

В жизни корпораций означает предоставление официального одобрения чего-либо. например: «Вице-президент по инжинирингу благословил наш проект WebCruncher». Аналогично в Perl это слово означает предоставление официального одобрения на то, чтобы *объект ссылки* мог функционировать как *объект*, например объект WebCruncher. См. описание функции `bless` в главе 27.

остроконечник (обратный порядок следования байтов) — little-endian

Из Свифта: тот, кто разбивает яйцо с острого конца. Употребляется также в отношении компьютеров, хранящих младший *байт* слова в младшем адресе, а старший байт – в старшем. Часто считается, что такие компьютеры превосходят «тупоконечные» машины. См. также *тупоконечник (прямой порядок следования байтов)*.

отображение регистра — casemapping

Процесс преобразования символов строки в один из четырех регистров, поддерживаемых Юникодом; в Perl реализуется функциями `fc`, `lc`, `ucfirst` и `uc`.

отправлять — dispatch

Посылать что-либо в правильное место. Часто употребляется метафорически, означая передачу программного управления в точку, выбранную алгоритмически, часто путем поиска в таблице ссылок на *функции* или, в случае *методов* объекта, при обходе дерева наследования в поисках

самого специализированного определения метода.

охватывающий оператор – circumfix operator

Оператор, охватывающий свой *операнд*, подобно оператору угловых скобок, или круглым скобкам, или объятиям.

очистка буфера – flush

Действие, состоящее в опустошении *буфера*, часто при его переполнении.

ошибка – error

См. *исключение* или *фатальная ошибка*.

пакет – package

Пространство имен для глобальных *переменных*, *подпрограмм* и т. п., предназначенное, чтобы хранить их отдельно от одноименных *символов* в других пространствах имен. В некотором смысле только пакет является глобальным, так как символы в таблице символов пакета доступны за его пределами только через упоминание пакета. Но в другом смысле все символы пакета являются глобальными – это просто хорошо организованные глобальные символы.

память – memory

Всегда означает оперативную память, а не диск. Осложняет дело то обстоятельство, что на машине может быть реализована *виртуальная память*, т.е. машина будет делать вид, что у нее больше памяти, чем есть на самом деле, используя дисковое пространство для хранения неактивных страниц памяти. При этом может показаться, что памяти несколько больше, чем есть на самом деле, но виртуальная память не является заменой настоящей памяти. Самое хорошее, что можно сказать о виртуальной памяти, – она позволяет производительности падать постепенно, а не внезапно, когда кончается память настоящая. Но программа может умереть, когда закончится и виртуальная память тоже, если раньше вы не загоните свой диск до смерти.

память совместного доступа – shared memory

Участок *памяти*, к которому имеют доступ два разных *процесса*, которые в ином случае не имели бы возможности видеть память друг друга.

параметр – parameter

См. *аргумент*.

патч – patch

«Заплатка» – исправление путем наложения. На языке хакеров – список различий между двумя версиями программы, который можно применить с помощью программы *patch*(1), чтобы исправить ошибку или обновить старую версию.

перевод строки – newline

Одиночный символ, представляющий конец строки, со значением ASCII «восьмеричное 012» в UNIX (но 015 в Mac) и представляемый метазнаком `\n` в строках Perl. В операционных системах Windows, при выводе в текстовые файлы и в некоторые физические устройства, такие как терминалы, одиночный символ перевода строки автоматически транслируется библиотекой C в перевод строки и возврат каретки, но обычно никакой трансляции не производится.

перегрузка – overloading

Сообщение дополнительного смысла символу или конструкции. Фактически любые языки в той или иной мере производят перегрузку, поскольку человек умеет определять смысл из *контекста*.

перегрузка операторов – operator overloading

Вид *перегрузки*, который можно производить для встроенных *операторов*, чтобы заставить их работать с объектами, как если бы они были обычными скалярными значениями, но с фактической семантикой, предоставляемой классом объекта. Перегрузка устанавливается *прагмой* `overload` – см. главу 13.

переключатель – switch statement

Прием программирования, позволяющий вычислить *выражение* и исходя из его значения осуществить переход к одному из нескольких участков кода. Называется также «case structure», в честь одноименной конструкции языка Pascal. Большинство

переключателей в Perl записываются посредством given. См. «Инструкция given» в главе 4.

переменная – variable

Именованный адрес ячейки памяти, способной хранить *значение* любого типа, необходимое вашей программе.

переменная окружения – environment variable

Механизм, посредством которого некий агент более высокого уровня, например пользователь, может сообщить о своих предпочтениях будущему потомку (порожденному *процессу*, внучатому процессу и т.д.). Каждая переменная среды представляет собой пару *ключ/значение*, как элемент в *хеше*.

переменная экземпляра – instance variable

Атрибут объекта; данные, хранимые в конкретном объекте, а не в классе.

переменные захвата – capture variables

Переменные, такие как $1 и $2, а также %+ и %-, которые хранят текст, захваченный при поиске по шаблону. См. главу 5.

переносимый – portable

Когда-то относилось к коду на C, который можно было компилировать в любой из систем BSD и SysV. В общем, код, который легко преобразовать для выполнения на другой *платформе*, где «легко» можно определить любым способом, что обычно и происходит. Все можно считать переносимым, если хорошенько постараться, например передвижной дом или Лондонский мост.

песочница – sandbox

Огороженная область, происходящее в которой не затрагивает окружающее пространство. Вы позволяете детям играть и не пускаете их на дорогу. См. главу 20.

плавающая запятая – floating point

Способ хранения чисел в «экспоненциальной записи», когда точность представления числа не зависит от того, насколько оно велико (десятичный разделитель – точка или запятая – «плавают»). Perl производит работу с числами с плавающей запятой, если *целых чисел* оказывается недостаточно. Числа с плавающей запятой являются приближением действительных чисел.

платформа – platform

Программно-аппаратный контекст, в котором выполняется программа. Программа, написанная на языке, зависящем от платформы, может слететь, если изменить машину или операционную систему, или библиотеку, или компилятор, или системную конфигурацию. Интерпретатор *perl* должен *компилироваться* отдельно для каждой платформы, потому что он реализован на C, но программы, написанные на языке Perl, в значительной мере платформонезависимы.

по умолчанию – default

Значение, выбранное за вас, если вы не указали собственное.

побочные эффекты – side effects

Нечто дополнительное, что происходит при вычислении *выражения*. В настоящее время это может касаться почти чего угодно. Например, простой оператор присваивания имеет «побочный эффект» – присваивание значения переменной. (А вы думали, что присваивание значения было вашей главной целью!) Аналогично присваивание значения специальной переменной $| ($AUTOFLUSH) имеет побочный эффект, заключающийся в принудительном очищении буфера после каждой операции write или print с выбранным в данный момент дескриптором файла.

подкласс – subclass

См. *производный класс*.

подпрограмма – subroutine

Доступный по имени или иным образом участок программы, который можно вызвать из любого места программы, чтобы выполнить некоторую подзадачу программы. Подпрограмма часто имеет параметры, позволяющие менять ее поведение в рамках определенного круга задач в зависимости от входных *аргументов*. Если подпрограмма возвращает осмысленное *значение*, ее называют также *функцией*.

подстановка – substitution

Изменение частей строки через оператор s///. (Мы избегаем использования этого термина в значении *интерполяции переменных*.)

подстрока – substring

Часть строки, начинающаяся с некоторой позиции *символа* (*смещения*) и охватывающая некоторое число символов.

подход ящика с инструментами – toolbox approach

Идея состоит в том, что с полным набором простых взаимодополняющих инструментов можно создать почти все, что нужно. Это хорошо, когда надо собрать трехколесный велосипед, но если мы создаем что-то необычное[1], нам понадобится собственный механический цех, в котором можно создавать специальные инструменты. Perl – своего рода механический цех.

подшаблон – subpattern

Компонент шаблона *регулярного выражения*.

подшаблон кода – code subpattern

Подшаблон *регулярного выражения*, действительной задачей которого является выполнение некоторого кода Perl. Так ведут себя, к примеру, подшаблоны (?{...}) и (??{...}).

***позиция косвенного объекта* – indirect object slot**

Синтаксическая позиция между вызовом метода и его аргументами при использовании синтаксиса вызова косвенного объекта. (Позиция отличается отсутствием запятой между ней и следующим аргументом.) В следующем примере STDERR является позицией косвенного объекта:

```
print STDERR "Awake! Awake!
    Fear, Fire, Foes! Awake!\n";
```

поиск по шаблону – pattern matching

Означает взять шаблон, обычно являющийся *регулярным выражением*, и попытаться различными способами наложить его на строку, чтобы найти соответствие. Часто применяется, чтобы найти в файле интересующие участки данных.

поиск с возвратом – backtracking

Обычай говорить: «Если бы начать все сначала, я бы все сделал иначе» – и затем действительно возвращаться и делать все по-другому. На математическом языке это означает возврат из безрезультатной рекурсии по дереву возможностей. Perl осуществляет поиск с возвратами при попытке найти соответствие шаблону с помощью *регулярного выражения*, когда из прежних попыток ничего не вышло. См. раздел «Маленький Механизм, который /(не)? может/» в главе 5.

поле – field

Фрагмент числовых или строковых данных, являющийся частью более длинной *строки* или *записи*. Поля переменной ширины обычно перемежаются *разделителями* (а извлекаются посредством split), тогда как поля фиксированной ширины обычно находятся в фиксированных позициях (и тогда нужно использовать unpack). *Переменные экземпляра* тоже называют «полями».

---

[1] В оригинале говорится о приборе defranishizing comboflux regurgalator – разупрощающем сложнопоточном регургаляторе; слово «регургалятор» раньше было длиннее и читалось «регуругалятор», обозначая некое приспособление, при помощи которого йоги могли очень долго вдыхать (inhale) продукты сухой возгонки сложных растительных смесей (в замкнутом цикле, это важно), достигая в результате нирваны и просветления. Впоследствии секрет этого приспособления был утерян, йоги больше не достигают просветления, а лишь впадают в нирвану (внешне это напоминает спячку, только человек при этом сидит и раскачивается, время от времени что-то тихонько напевая). Но слово не исчезло. Буква «у» редуцировалась и термин обозначает теперь довольно несложный, хотя и секретный прибор (регургалятор), состоящий на вооружении армии США. – *Прим. ред.*

полиморфизм — polymorphism

Идея заключается в том, что можно приказать *объекту* выполнить некоторое родовое действие, и объекты интерпретируют команду различными способами, в зависимости от своих типов. [От греческого πολυ- + μορφή – много форм.]

полубайт — nybble

Половина *байта*, эквивалентная одной шестнадцатеричной цифре и занимающая четыре *бита*.

поразрядный сдвиг — bit shift

Перемещение битов влево или вправо в машинном слове, которое приводит к умножению или делению на степень двойки.

порт — port

Часть адреса TCP или UDP, которая направляет пакеты нужному процессу, найдя нужную машину, – нечто вроде добавочного номера, который мы сообщаем, дозвонившись до коммутатора компании. Кроме того, означает результат преобразования кода для платформы, отличной от избранной первоначально.

порядковое — ordinal

Целочисленное значение абстрактного символа. То же, что *код символа*.

порядок поиска методов — method resolution order

Порядок обхода записей в @INC. По умолчанию выполняется двойной поиск методов в глубину, сначала среди определенных методов, а потом в блоке AUTOLOAD. Однако Perl позволяет изменять порядок с помощью mro.

порядок следования байтов — endian

См. *тупоконечник (прямой порядок следования байтов)* и *остроконечник (обратный порядок следования байтов)*.

поступательный поиск — progressive matching

Поиск по шаблону, продолжающий работу с того места, где остановился в предыдущий раз.

постфиксный оператор — postfix

Оператор, следующий за своим операндом, например $x++.

поток — stream

Поток данных в процесс или из процесса как устойчивая последовательность байтов или символов без видимости разбиения на пакеты. Это своего рода *интерфейс* – соответствующая *реализация* вполне может разбивать данные на пакеты для доставки, но это скрыто от пользователя.

поток выполнения — thread

Подобен ответвленному процессу, но без присущей fork защиты памяти. Поток выполнения – более легковесный, чем полноценный процесс, так как процесс может иметь несколько потоков, выполняемых в нем, и все они претендуют на пространство памяти одного и того же процесса, если только не предпринимаются шаги для защиты потоков друг от друга.

право первого — first-come

Первый, выгрузивший на сервер *PAUSE* «пространство имен», автоматически становится *главным хранителем* этого пространства имен. Привилегии, полученные автоматически по «праву первого», отличаются от привилегий *главного хранителя*, которые могут передаваться другим лицам.

правостороннее значение — rvalue

Значение, которое можно поместить в правой части присваивания. См. также *левостороннее значение*.

прагма (директива) — pragma

Стандартный модуль, практические советы и предложения которого получаются (и, возможно, игнорируются) на этапе компиляции. Имена прагм задаются целиком в нижнем регистре.

предупреждение — warning

Сообщение, выводимое в поток STDERR с целью показать, что есть проблемы, но не настолько серьезные, чтобы прекратить работу. См. warn в главе 27 и прагму warnings в главе 29.

преобразование в строку – stringification

Процесс создания представления абстрактного объекта в виде *строки*.

препроцессинг, предварительная обработка – preprocessing

То, что некоторый вспомогательный *процесс* осуществил для преобразования входных данных в форму, более пригодную для текущего процесса. Часто выполняется через входной *канал*. См. также *препроцессор C*.

препроцессор C – C preprocessor

Первый проход типичного компилятора C, в котором обрабатываются строки, начинающиеся символом #, для условной компиляции и макроопределений, а также осуществляются различные манипуляции с текстом программы на основе текущих определений. Известен также как *cpp*(1).

префиксный оператор – prefix

Оператор, предшествующий своему *операнду*, например ++$x.

приведение типа – type casting

Преобразование данных из одного типа в другой. Язык C допускает это. Perl не нуждается в этом. И не хочет этого.

привязка – bind

Назначение конкретного *сетевого адреса сокету*.

приложение – application

Разновидность больших и сложных *программ* с замысловатым названием, чтобы люди не замечали, что пользуются программой.

приоритет – precedence

Правила поведения, которыми, в отсутствие других указаний, определяется, что должно произойти раньше. Например, при отсутствии скобок умножение всегда выполняется раньше сложения.

присваивание – assignment

Оператор, цель которого – изменять значение *переменной*.

присваивания, оператор – assignment operator

Обычное *присваивание*, либо составной *оператор*, образованный из обычного присваивания и некоторого другого оператора, который изменяет значение переменной по месту, т.е. относительно ее прежнего значения. Например $a += 2 прибавляет 2 к $a.

пробельный символ – whitespace

Символ, который перемещает курсор, но ничего на экран не выводит. Обычно относится к любому из символов: пробел, табуляция, перевод строки, возврат каретки, перевод формата. В Юникоде соответствует многим другим символам, включая NO-BREAK SPACE.

проверка меченых данных – taint checks

Механизм, позволяющий Perl следить за перемещением внешних данных в пределах вашей программы и запрещать их использование в системных командах.

программа – program

См. *сценарий*.

программное обеспечение с открытым исходным кодом – open source software

Программы, для которых можно бесплатно получить исходный код и свободно распространять его дальше, без коммерческих условий. Более точное определение можно найти в документе *http://www.opensource.org/osd.html*.

продолжение – continuation

Работа с одной или более физическими *строками* как с одной логической строкой. Строки *Makefile* продолжаются с помощью обратной косой черты перед символом *перевода строки*. Почтовые заголовки по определению RFC 822 продолжаются с помощью пробела или табуляции *после* перевода строки. В целом для строк Perl не требуется какой-либо знак продолжения, потому что *пробельный символ* (в том числе символ перевода строки) с радостью игнорируется. Как правило.

производный класс — derived class

Класс, определяющий некоторые из своих *методов* через более общий класс, называемый *базовым классом*. Обратите внимание, что классы не разделяются исключительно на базовые или производные: класс может одновременно функционировать как производный и базовый, что, конечно, классно.

пространство имен — namespace

Домен имен. Нет необходимости беспокоиться, что имена из одного такого домена используются в другом. См. *пакет*.

протокол — protocol

В сетевом взаимодействии — согласованный способ передачи сообщений в обоих направлениях, который не должен слишком смутить ни одного из корреспондентов.

прототип — prototype

Необязательная часть объявления *подпрограммы*, сообщающая компилятору Perl, сколько и каких аргументов можно передать в качестве *фактических аргументов*, чтобы можно было писать вызовы подпрограмм, распознаются анализатором так же, как вызовы встроенных функций. (Или не распознаются, как может случиться.)

процедура — procedure

Подпрограмма.

процесс — process

Экземпляр выполняющейся программы. В многозадачных системах типа UNIX два или более отдельных процесса могут одновременно и независимо выполнять одну и ту же программу — на практике для обеспечения такой счастливой жизни предназначена функция `fork`. В других операционных системах процессы иногда называются «threads», «tasks», «jobs» (потоки, задачи, задания), причем различия в названиях могут подразумевать небольшие различия в значениях.

псевдолитерал — pseudoliteral

Оператор, похожий на литерал, например оператор захвата вывода `` `command` ``.

псевдоним — alias

Прозвище чего-либо, во всех отношениях ведущее себя так же, как если бы вместо него использовалось исходное имя. Временные псевдонимы неявно создаются в переменной цикла для `foreach`, в переменной `$_` для операторов `map` или `grep`, в `$a` и `$b` во время выполнения функции сравнения в `sort` и в каждом элементе `@_` для *фактических аргументов* вызова подпрограммы. Постоянные псевдонимы явно создаются в *пакетах* путем *импортирования* символов или присваивания переменным *typeglob*. Псевдонимы с лексической областью видимости для переменных пакетов явно создаются объявлением `our`.

псевдофункция — pseudofunction

Конструкция, которая иногда выглядит как функция, но в действительности ею не является. Обычно относится к модификаторам *левосторонних значений*, например `my`, модификаторам контекста, например `scalar`, и к конструкциям для выбора собственных кавычек `q//`, `qq//`, `qx//`, `qw//`, `qr//`, `m//`, `s///`, `y///` и `tr///`.

псевдохеш — pseudohash

Прежде — ссылка на массив, первый элемент которого содержит ссылку на хеш. С псевдохешем можно работать и как со ссылкой на массив, и как со ссылкой на хеш. Ныне псевдохеши не поддерживаются.

пустая строка — null string

Строка, не содержащая символов; не путать со строкой, содержащей *нулевой символ*, которая имеет положительную длину и является *истинной*.

пустой контекст — void context

Вид *скалярного контекста*, в котором не предполагается возврат выражением какого-либо *значения*, а вычисление его осуществляется только ради *побочных эффектов*.

пустой список — null list

Списочное значение, содержащее ноль элементов, представляемое в Perl как `()`.

путь, маршрут – pathname

Полностью квалифицированное имя файла, например */usr/bin/perl*. Иногда путают с PATH.

рабочий каталог – working directory

Текущий *каталог*, от которого *операционная система* прокладывает относительные маршруты. Операционная система определяет текущий каталог как тот, который был указан ей посредством команды chdir, или как то место, в котором находился ваш родительский *процесс* при вашем появлении на свет.

разбиение на лексемы – tokenizing

Расщепление текста программы на *лексемы*. Другое название – «lexing».

разделитель – separator

Символ или *строка*, позволяющие разграничить две окружающие их строки. С разделителями работает функция split. Не путать с *ограничителями* и *терминаторами*. Союз «и» в предыдущем предложении был разделителем двух альтернатив.

разрушать – destroy

Освобождать память *объекта ссылки* (сначала вызывая метод DESTROY, если он есть).

разрыв строки – linebreak

Графема – возврат каретки, за которым следует перевод строки, либо любой символ со *свойством символа* Юникода Vertical Space.

разыменовывать – dereference

Причудливый термин вычислительной техники, означающий «следовать за *ссылкой* к тому, на что она указывает». Приставка «раз-» относится к тому обстоятельству, что вы удаляете один уровень косвенности.

разыменовывающий символ – funny character

Некто, похожий на Ларри или одного из его эксцентричных друзей (funny character – это и «забавный персонаж»). Относится также к необычным префиксам, которыми Perl требует помечать свои переменные-существительные.

расширение – extension

Модуль Perl, загружающий также скомпилированный код на C или C++. В более широком смысле – любая экспериментальная возможность, которая интегрируется в Perl в результате *компиляции*; например, поддержка многозадачности.

реализация – implementation

То, как фрагмент кода фактически выполняет свою задачу. Пользователи кода не должны полагаться, что детали реализации остаются постоянными, если только речь не идет об общедоступном *интерфейсе*.

регистр – case

Свойство некоторых символов. Первоначально наборщики хранили литеры с прописными буквами в верхнем ящике (case), а строчные – в нижнем. Юникод распознает три регистра: *нижний* (*свойство символа* \p{lower}), *заглавный* (\p{title}) и *верхний* (\p{upper}). Четвертый регистр называется *сверткой* и не является самостоятельным регистром, но используется в реализации свертки регистра. Не все буквы имеют регистр, и некоторые небуквенные символы имеют регистр.

регулярное выражение – regular expression

Объект, имеющий несколько интерпретаций, как слон. Для ученого в области вычислительных наук это грамматика небольшого языка, в котором одни строки допустимы, а другие – нет. Для обычных людей это шаблон, который можно использовать, чтобы найти то, что нужно, если оно иногда меняется. Регулярные выражения Perl далеки от регулярности в теоретическом смысле, но при регулярном применении они работают вполне хорошо. Регулярное выражение /Oh s.*t./ соответствует таким строкам, как "Oh say can you see by the dawn's early light" и "Oh sit! ". См. главу 5.

регулярный файл – regular file

Файл, не являющийся *каталогом*, *устройством*, именованным *каналом*, *сокетом* или *символической ссылкой*. Perl идентифицирует регулярные файлы с помощью оператора проверки файла -f. Иногда они называются «обычными» (plain) файлами.

режим – mode

В контексте системного вызова *stat*(2) указывает на поле, содержащее *биты разрешений* и тип файла.

режим меченых данных – taint mode

Включается при запуске с ключом -T, обеспечивает маркировку всех внешних данных, с целью отследить движение данных в программе и воспрепятствовать их использованию в системных командах. См. главу 20.

рекурсия – recursion

Искусство определить что-либо (хотя бы частично), ссылаясь на самое себя. В словарях это порочная практика, но в компьютерных программах часто работает хорошо, если следить, чтобы не образовывалась бесконечная рекурсия, которая сходна с бесконечным циклом, но влечет более зрелищный отказ.

ретроспективная проверка – lookbehind

Утверждение, которое заглядывает в строку левее текущего найденного соответствия.

родительский класс – parent class

См. *базовый класс*.

роль – role

Название конкретного набора поведений. Роли позволяют расширять поведение класса, не прибегая к наследованию.

самое левое самое длинное – leftmost longest

Стремление механизма *регулярных выражений* найти самое левое соответствие шаблону; затем, исходя из позиции, в которой найдено соответствие, стремление найти самое длинное соответствие (в предположении использования *жадного* квантификатора). Подробности об этом предмете см. в главе 5.

самооживление – autovivification

Греко-романское слово, означающее «оживить самого себя». В Perl адреса хранения (*левосторонние значения*) самопроизвольно создаются по мере необходимости, в том числе создаются значения *жестких ссылок*, указывающих на следующий уровень хранения. Присваивание `$a[5][5][5][5][5] = "quintet"` потенциально создает пять адресов хранения скаляров плюс четыре ссылки (в первых четырех адресах скаляров), указывающие на четыре новых анонимных массива (для хранения четырех последних адресов скаляров). Но смысл самооживления в том, что программисту не надо об этом беспокоиться.

сборка мусора – garbage collection

Неправильно названная функция – ее следовало назвать «расчет на то, что мать уберет после вас». Строго говоря, Perl этого не делает, а полагается на механизм подсчета ссылок, который приведет все в порядок. Однако мы редко выражаемся строго и будем часто называть схему подсчета ссылок формой уборки мусора. (Может быть, вас утешит, что по завершении работы интерпретатора запускается «настоящий» сборщик мусора, который обеспечит уборку, если вы неряшливо обращались с циклическими ссылками и т. п.)

свертка регистра – casefolding

Используется для сравнения или сопоставления строк без учета регистра символов. В Perl свертка регистра реализуется модификатором /i шаблонов, функцией fc и управляющей последовательностью \F в интерполируемых строках.

свертка регистра – foldcase

Регистр, используемый в Юникоде для сравнения или сопоставления без учета регистра символов. Сравнение в нижнем, заглавном или верхнем регистре не всегда дает надежные результаты из-за сложной, «один-ко-многим», схемы отображения регистров символов в Юникоде. Свертка регистра – это разновидность *нижнего регистра* (с использованием для некоторых кодов символов формы *нормализации* с декомпозицией), созданная специально для решения этой проблемы.

свойство – property

См. *переменная экземпляра* или *свойство символа*.

свойство символа – character property

Предопределенный класс *символов*, соответствующих метазнаку \p или \P. Для *Юникода* определены сотни стандартных свойств для каждого кодового пункта, а Perl определяет ряд дополнительных свойств.

связка – bundle

Группа взаимосвязанных модулей в *CPAN*. (Иногда также означает группу ключей командной строки, объединенных в один *кластер ключей*.)

связующий язык – glue language

Язык (например, Perl), который удобен для соединения вместе вещей, изначально для соединения не предназначенных.

связь – tie

Связь между магической переменной и классом ее реализации. См. функцию tie в главе 27 и главу 14.

сдвиг влево – left shift

Поразрядный сдвиг, который умножает число на некоторую степень двойки.

сдвиг вправо – right shift

Поразрядный сдвиг, который делит число на некоторую степень двойки.

семафор – semaphore

Вид блокировки, предотвращающий одновременное использование одних и тех же ресурсов несколькими *потоками выполнения* или *процессами*.

сервер – server

В сетевом взаимодействии – *процесс*, предоставляющий *услугу* или просто ожидающий в известном месте *клиентов*, которые обращаются к нему за услугами.

сериализация – serialization

Реорганизация сложной *структуры данных* в линейный порядок с целью сохранения в виде *строки* в дисковом файле или базе данных либо с целью передачи через *канал*. Другое название – маршалинг (marshalling).

сетевой адрес – network address

Важнейший атрибут сокета, подобный телефонному номеру. Обычно IP-адрес. См. также *порт*.

сигнал – signal

Гром среди ясного неба; событие, порожденное операционной системой, когда вы меньше всего его ожидали.

символ – character

Наименьший строительный элемент строки. Компьютеры хранят символы как целые числа, но Perl позволяет работать с ними как с текстом. Целое число, представляющее символ, нзывается *кодом символа*.

символ – symbol

В целом любая *лексема* или *метазнак*. Часто употребляется в более узком смысле для обозначения имени, которое можно найти в *таблице символов*.

символическая ссылка – symbolic link

Альтернативное имя файла, указывающее на действительное *имя файла*, которое, в свою очередь, указывает на действительный *файл*. Когда *операционная система* пытается анализировать *маршрут*, содержащий символическую ссылку, она просто подставляет новое имя и продолжает анализ.

символическая ссылка – symbolic reference

Переменная, значением которой является имя другой переменной или подпрограммы. Путем *разыменования* первой переменной можно получить вторую. Символические ссылки недопустимы при использовании директивы use strict "refs".

символический отладчик – symbolic debugger

Программа, позволяющая *выполнять* другую программу в пошаговом режиме, время от времени останавливаясь и выводя данные, чтобы посмотреть, не пошло ли что-нибудь не так, и если да, то что. Слово «символический» означает, что с отладчиком можно разговаривать, используя те же символы, с которыми была написана ваша программа.

синтаксис – syntax

От греческого «σύνταξις». Порядок, в соответствии с которым вещи (в особенности символы) располагаются совместно.

синтаксический анализ – parsing

Тонкое, но иногда жестокое искусство преобразования вашей, возможно, плохо сформированной программы в допустимое синтаксическое дерево.

синтаксический сахар – syntactic sugar

Альтернативный способ более простого написания; сокращение.

синтаксическое дерево – syntax tree

Внутреннее представление программы, при котором *конструкции* более низкого уровня свисают из заключающих их конструкций более высокого уровня.

синхронный – synchronous

Способ программирования, когда события определяются как организованная последовательность; т.е. когда события происходят одно за другим, а не в одно и то же время.

системный вызов – syscall

Вызов *функции*, непосредственно обращенный к *операционной системе*. Многие важные подпрограммы и функции, которые мы используем, не являются непосредственными системными вызовами, но стоят на один или несколько уровней выше, чем системные вызовы. В целом, программистам Perl не нужно беспокоиться об этих различиях. Однако человек, знающий, какие из функций Perl в действительности являются системными вызовами, сможет предсказать, какая из них установит переменную `$!` (`$ERRNO`) в случае отказа. К сожалению, начинающие программисты часто ошибочно называют «системным вызовом» вызов функции `system`, которая фактически выполняет множество системных вызовов. Чтобы избежать путаницы, мы почти всегда говорим «системный вызов», когда имеется в виду то, что можно косвенно вызвать через функцию `syscall`, и никогда в отношении того, что следует вызывать функцией `system`.

скаляр – scalar

Простое отдельное значение; число, *строка* или *ссылка*.

скалярная переменная – scalar variable

Переменная с префиксом `$`, содержащая одно значение.

скалярное значение – scalar value

Значение, являющееся скаляром (ср. *список*).

скалярный контекст – scalar context

Ситуация, когда окружение (вызывающий код) ожидает, что *выражение* вернет единственное *значение*, а не *список*. См. также *контекст* и *списочный контекст*. Скалярный контекст иногда накладывает дополнительные ограничения на возвращаемое значение – см. *строковый контекст* и *числовой контекст*. Иногда мы говорим о *булевом контексте* внутри условных операторов, но он не накладывает дополнительных ограничений, потому что любое скалярное значение, число или *строка* уже является истинным или ложным.

скалярный литерал – scalar literal

Число или *строка* в кавычках – фактическое значение в программе (ср. *переменная*).

слабая ссылка – weak reference

Ссылка, не учитываемая механизмом подсчета ссылок. Когда исчезает последняя нормальная ссылка на данные, они утилизируются сборщиком мусора. Слабые ссылки могут использоваться для создания циклических ссылок, не препятствующих утилизации ненужных данных, как нормальные ссылки.

слово – word

На обычном компьютерном языке – фрагмент данных такого размера, который наиболее эффективно обрабатывается компьютером, обычно 32 бита плюс-минус несколько степеней двойки. В культуре Perl чаще относится к буквенно-цифровому *идентификатору* (включая символы подчеркивания) или к строке *символов*, отличных от пробельных, ограниченной по кра-

ям пробельными символами или границей строки.

смертник — mortal

Временное значение, которое должно умереть, когда закончится текущая команда.

смещение — offset

Количество элементов, которое следует пропустить, перемещаясь от начала строки или массива в заданную позицию. Поэтому минимальное смещение равно нулю, а не единице, поскольку не надо ничего пропускать, чтобы добраться до первого элемента.

соединение — connection

В телефонии это создание временной электрической цепи между аппаратами вызывающего и вызываемого абонентов. В сетевом взаимодействии это такая же временная цепь между *клиентом* и *сервером*.

сокет — socket

Конечная точка сетевого соединения между несколькими *процессами*, которая действует во многом сходно с телефонной кабинкой или почтовым ящиком. Самое важное в сокете — его *сетевой адрес* (как номер телефона). Различные виды сокетов имеют различные виды адресов: некоторые похожи на телефонные номера, а некоторые — нет.

соответствие — matching

См. *поиск по шаблону*.

сортирующая последовательность — collating sequence

Порядок, согласно которому сортируются *символы*. Им руководствуются, например, программы сортировки строк, решая, в какое место данного глоссария поместить «порядок сравнения».

состояние гонки — race condition

Состояние гонки возникает, когда результат не связанных между собой событий зависит от порядка, в котором они наступают, но этот порядок невозможно гарантировать из-за недетерминированности временных эффектов. Если две или более программ или частей одной программы пытаются пройти через ту же самую серию событий, одна может прервать работу другой. Это можно использовать для поиска уязвимостей и создания *эксплойтов*.

со-хранитель — co-maintainer

Человек, обладающий правом индексировать *пространство имен* в *PAUSE*. Любой может выгружать исходные тексты в любое пространство имен, но только основной хранитель и со-хранители могут вносить свой вклад, доступный для индексирования.

СПИСОК — LIST

Синтаксическая конструкция, представляющая разделенный запятыми список выражений, производящих при вычислении *списочное значение*. Каждое *выражение* в `LIST` вычисляется в *списочном контексте* и интерполируется в списочное значение.

список — list

Упорядоченное множество скалярных значений.

списочное значение — list value

Безымянный список временных скалярных величин, который может передаваться в программе от любой функции, генерирующей список, любой функции или конструкции, предоставляющей *списочный контекст*.

списочный контекст — list context

Ситуация, когда окружение (вызывающий код) ожидает, что выражение вернет список значений, а не отдельное значение. Функции, которым требуется *список* аргументов, сообщают этим аргументам, что они должны создать списочное значение. См. также *контекст*.

списочный оператор — list operator

Оператор, делающий что-то со списком значений, например `join` или `grep`. Обычно используется для именованных встроенных операторов (таких как `print`, `unlink` и `system`), в которых список *аргументов* можно не заключать в круглые скобки.

среда — environment

Совокупность *переменных среды*, унаследованных процессом от родителя. Доступ осуществляется через %ENV.

срез — slice

Произвольное число *элементов*, отобранных из *списка*, *массива* или *хеша*.

ссылка — reference

Место, где находится указатель на информацию, хранящуюся где-то еще. (См. *косвенность*.) Ссылки бывают двух видов: *символические ссылки* и *жесткие ссылки*.

ссылка; компоновать — link

При использовании в качестве существительного – имя в *каталоге*, представляющее файл. На заданный файл может быть несколько ссылок. Это похоже на включение в телефонный справочник одного и того же номера телефона под разными именами. В качестве глагола означает разрешение неразрешенных символов частично *скомпилированного* файла в (почти) выполняемый образ. Компоновка может быть статической или динамической, что не имеет никакого отношения к статической или динамической области видимости.

ссылка на найденный текст — backreference

Подстрока, *захваченная* подшаблоном в простых скобках в *регулярном выражении*. Десятичные числа с обратной косой чертой перед ними (\1, \2 и т.д.) в том же шаблоне ссылаются на соответствующий подшаблон в текущем поиске. Вне шаблона нумерованные переменные ($1, $2 и т.д.) продолжают ссылаться на те же значения, пока этот шаблон является последним успешно сопоставленным шаблоном в текущей *динамической области видимости*.

стандартная библиотека — Standard Library

Все, что распространяется в официальном дистрибутиве *perl*. Некоторые производители собирают свои версии *perl*, изменяя состав дистрибутива, исключая какие-то элементы или включая дополнительные. См. также *двойная жизнь*.

стандартное устройство ввода — standard input

Входной *поток* данных программы по умолчанию, который, если это возможно, не должен обращать внимания на то, откуда поступают данные. Представлен в программе на Perl *дескриптором файла* STDIN.

стандартное устройство вывода — standard output

Выходной *поток* данных программы по умолчанию, который, если это возможно, не должен обращать внимания на то, куда отправляются данные. Представлен в программе на Perl *дескриптором файла* STDOUT.

стандартное устройство ошибок — standard error

Стандартный выходной *поток* данных для неприятных замечаний, которые не следует записывать в *стандартное устройство вывода*. В программе на Perl представлен *дескриптором файла* STDERR. Можно явно использовать этот поток, а встроенные функции die и warn пишут в стандартное устройство вывода ошибок автоматически (если вывод не перехватывается и не обрабатывается каким-то иным способом).

стандартный — standard

Включенный в официальный дистрибутив Perl: стандартный модуль, стандартное инструментальное средство, стандартная *страница руководства* Perl.

стандартный ввод/вывод — standard I/O

Стандартная библиотека С для осуществления *буферизованного* ввода и вывода в *операционную систему*. («Стандарт» стандартного ввода/вывода лишь частично связан со «стандартами» стандартных устройств ввода/вывода.) В целом Perl полагается на реализацию стандартного ввода/вывода, предоставляемую операционной системой, поэтому характеристики буферизации программы Perl на одной машине могут не вполне совпадать с подобными характеристиками на другой машине. Обычно это влияет только на эффективность, а не на семантику. Если ваш пакет стандартного ввода/вывода осуществляет только блочную буферизацию, а вам нужно *очищать*

буфер чаще, установите переменную $| в истинное значение.

статическая область видимости – static scoping

Не поддерживается. См. *лексическая область видимости*.

статическая переменная – static variable

Не поддерживается. Используйте *лексическую переменную* в области видимости, большей, чем ваша *подпрограмма*, или объявите ее с помощью ключевого слова state вместо my.

статический – static

Медленно меняющийся относительно чего-то другого. (К сожалению, все является относительно стабильным в сравнении с чем-то еще, за исключением некоторых элементарных частиц, в отношении которых нельзя говорить уверенно.) В компьютерах, где все должно изменяться быстро, термин «статический» имеет презрительный оттенок, указывая на не вполне функциональные *переменную*, *подпрограмму* или *метод*. В культуре Perl этого слова вежливо избегают.

статический метод – static method

Не поддерживается. См. *метод класса*.

статус – status

Значение, возвращаемое родительскому *процессу*, когда один из порожденных им процессов завершается. Это значение помещается в специальную переменную $?. Старшие восемь *разрядов* содержат код завершения процесса, а младшие восемь определяют сигнал (если он был), завершивший процесс. В системах UNIX это значение статуса совпадает со словом статуса, возвращаемым *wait*(2). См. system в главе 27.

статус завершения – exit status

См. *статус*.

стек – stack

Устройство, на вершину которого можно что-то помещать, а затем извлекать в обратном порядке. См. *LIFO*.

страница руководства – manpage

«Страница» из руководства, доступ к которой обычно осуществляется через команду *man*(1). Страница руководства содержит разделы SYNOPSIS (аннотация), DESCRIPTION (описание), BUGS (список известных ошибок) и т.д. и обычно длиннее печатной страницы. Страницы руководства документируют *команды*, *системные вызовы*, *библиотечные функции*, *устройства*, *протоколы*, *файлы* и т.п. В данной книге мы называем страницей руководства любую часть стандартной документации Perl (например, *perlop* или *perldelta*), независимо от того, в каком формате она установлена в конкретной системе.

строка – string

Последовательность символов, такая как «Она сказала !@#*&%@#*?!». Строка не обязательно должна быть полностью печатаемой.

строковый контекст – string context

Ситуация, когда окружение (вызывающий код) ожидает, что выражение вернет строку. См. также *контекст* и *числовой контекст*.

строчка – line

В UNIX это последовательность из нуля или более символов, отличных от перевода строки, завершающаяся символом *перевода строки*. На прочих платформах строки эмулируются библиотекой C, даже если операционная система имеет иные представления.

структура – structure

См. *структура данных*.

структура stat – stat structure

Особое место в памяти Perl, где хранятся данные о последнем файле, сведения о котором запрашивались.

структура данных – data structure

Способ объединения отдельных элементов данных, а также форма, которую данные совокупно образуют, – прямоугольная таблица или треугольное дерево.

суперпользователь — superuser

Лицо, которому *операционная система* позволяет делать почти все. Обычно это ваш системный администратор или некто, претендующий на роль системного администратора. В системах UNIX это пользователь *root*. В системах Windows это обычно пользователь *Administrator*.

сценарий — script

Текстовый файл, являющийся программой, предназначенной для непосредственного *выполнения*, а не *компиляции* в файл другого вида перед *выполнением*.

Кроме того, в контексте *Юникода* означает систему письменности языка или группы языков, например греческого, бенгальского или Tengwar.

таблица символов — symbol table

Место, где *компилятор* запоминает символы. Программа типа Perl должна каким-то образом запоминать все имена всех *переменных*, *дескрипторов файлов* и *подпрограмм*, которые были использованы. Она делает это, помещая имена в таблицу символов, которая реализована Perl с помощью *хеш-таблицы*. В каждом *пакете* присутствует собственная таблица символов, предоставляющая каждому пакету собственное *пространство имен*.

текст — text

Строка или *файл*, содержащие в основном печатаемые символы.

текущий выбранный канал вывода — currently selected output channel

Последний *дескриптор файла*, назначенный с помощью `select(FILEHANDLE)`; `STDOUT`, если не был выбран какой-либо дескриптор файла.

текущий пакет — current package

Пакет, в котором *скомпилирована* текущая команда. Ищите в обратном направлении в тексте программы, в текущей *лексической области видимости* или в охватывающих лексических областях, пока не найдете объявление пакета. Это и будет именем текущего пакета.

текущий рабочий каталог — current working directory

См. *рабочий каталог*.

тема — topic

То, над чем вы работаете. Конструкции, такие как `while(<>)`, `for`, `foreach` и `given`, определяют тему, по умолчанию присваивая значение переменной `$_`.

терм — term

От «терминальный», т.е. лист *синтаксического дерева*. То, что грамматически служит *операндом* для операторов выражения.

терминатор — terminator

Символ или строка, помечающие конец другой строки. Переменная `$/` содержит строку, завершающую операцию `readline`, которую `chomp` удаляет из конца строки. Не путать с *ограничителями* и *разделителями*. Точка в конце этого предложения является терминатором.

тернарный — ternary

Оператор, принимающий три *операнда*. Иногда произносится *trinary* (тринарный).

тест пустого подкласса — empty subclass test

Означает, что пустой *производный класс* должен вести себя, в точности как его *базовый класс*.

тип — type

См. *тип данных* и *класс*.

тип данных — data type

Множество допустимых значений, а также операции, умеющие обращаться с этими значениями. Например, для числового типа данных существует некое множество чисел, с которыми можно работать, а также различные математические операции, которые можно производить с числами, но которые не очень осмысленны, скажем, со строками. Для строк есть свои операции, например *конкатенация*. А для составных типов, образованных из некоторого количества меньших частей, обычно существуют операции образования и декомпозиции, а возможно, и реорганизации. *Объекты*, моделирующие предметы реального мира,

часто имеют операции, соответствующие их реальной деятельности. Например, если моделируется лифт, объект лифта должен иметь *метод* открыть_дверь().

типизованная лексическая переменная — typed lexical

Лексическая переменная, объявленная с типом *класса*: my Pony $bill.

транслитерировать — transliterate

Преобразовывать одно представление строки в другое путем отображения каждого символа исходной строки в соответствующий символ результирующей строки. Не путайте с трансляцией (переводом), например греческое слово πολύχρωμος транслитерируется в *polychromos*, но переводится как *многоцветный*. См. оператор tr/// в главе 5.

триггер — trigger

Событие, которое вызывает запуск *обработчика*.

троичный — trinary

Не система из трех звезд, а *оператор*, принимающий три операнда. Иногда пишется *ternary* (*тернарный*).

тупоконечник (прямой порядок следования байтов) — big-endian

Из Свифта: тот, кто разбивает яйцо с тупого конца. Употребляется также в отношении компьютеров, в которых старший байт слова хранится в младшем адресе, а младший — в старшем. Часто считается, что такие компьютеры превосходят «остроконечные» машины. См. также *остроконечник (обратный порядок следования байтов)*.

тыква — pumpkin

Условный «жезл», ходящий в сообществе Perl, которым владеет ведущий интегратор в какой-либо области разработок.

уборка — reaping

Последний ритуал, исполняемый родительским процессом в отношении почившего процесса-потомка, чтобы тот не превратился в *зомби*. См. вызовы функций wait и waitpid.

указатель — descriptor

См. *указатель файла*.

указатель — pointer

Переменная в языке типа C, содержащая точный адрес другого элемента в памяти. Perl обрабатывает указатели самостоятельно, и о них беспокоиться не следует. Вместо указателей используются символические указатели в виде *ключей* и имен *переменных* либо *жесткие ссылки*, которые указателями не являются (но действуют как указатели и фактически содержат в себе указатели).

указатель файла — file descriptor

Маленькое число, используемое *операционной системой* для слежения за открытыми *файлами*. Perl скрывает указатель файла внутри потока *стандартного ввода/вывода*, а затем прикрепляет поток к дескриптору файла.

унарный оператор — unary operator

Оператор с единственным *операндом*, такой как ! или chdir. Унарные операторы обычно являются префиксными, т.е. они предшествуют своему операнду. Операторы ++ и -- могут быть префиксными или постфиксными. (Их положение *изменяет* их смысл.)

усечение — truncating

Удаление из файла его содержимого — автоматически при открытии файла для записи или явным образом, посредством функции truncate.

условный оператор — conditional

Нечто сомнительное («iffy»). См. *булев контекст*.

услуга — service

Нечто, что мы выполняем для кого-то другого, чтобы удовлетворить его, например сообщаем ему время суток (или время, которое ему осталось). На некоторых машинах перечень доступных услуг можно получить вызовом функции getservent.

устройство — device

Аппаратная штуковина (типа диска, ленточного устройства, джойстика или мы-

ши), встроенная в компьютер, которую *операционная система* пытается заставить выглядеть как *файл* (или группа файлов). В UNIX эти ложные файлы обычно живут в каталоге */dev*.

утверждение — assertion

Составляющая *регулярного выражения*, которая должна быть истинной в случае соответствия шаблону, но не обязательно должна сама соответствовать каким-то символам. Часто используется в специализированном значении утверждения *нулевой ширины*.

фаза выполнения — run phase

Любой момент времени после того, как Perl запустит нашу основную программу. См. также *фаза компиляции*. Фаза выполнения в основном проходит на *этапе выполнения* (*runtime*), но может также проходить на *этапе компиляции* (*compile time*), когда выполняются операторы require, do *FILE* или eval *STRING*, либо когда подстановка использует модификатор /ee.

фаза компиляции — compile phase

Любой момент времени перед тем, как Perl начинает выполнять вашу основную программу. См. также *фаза выполнения*. Фаза компиляции в основном приходится на *этап компиляции*, но может также иметь место на *этапе выполнения*, когда вычисляются блоки BEGIN, объявления use или подвыражения констант. Код начального запуска и код импорта любого объявления use также выполняется в фазе компиляции.

файл — file

Именованная совокупность данных, обычно сохраненная на диске, в *каталоге файловой системы*. Примерно соответствует документу, если использовать канцелярские метафоры. В современных файловых системах файл может иметь несколько имен. Некоторые файлы, например каталоги и устройства, обладают особыми свойствами.

файловая система — filesystem

Набор *каталогов* и *файлов*, находящихся на разделе диска. Иногда называется «разделом» (partition) диска. Можно изменять имена файлов и даже перемещать файлы из каталога в каталог в пределах файловой системы без фактического перемещения самого файла, по крайней мере в системах UNIX.

фактические аргументы — actual arguments

Скалярные значения, передаваемые *функции* или *подпрограмме* при их вызове. Например, при вызове power("puff") строка "puff" представляет собой фактический аргумент. См. также *аргумент* и *формальные аргументы*.

фатальная ошибка — fatal error

Неперехваченное *исключение*, вызывающее завершение процесса после вывода сообщения в *стандартный поток ошибок*. Ошибки, возникающие внутри eval, не являются фатальными. Завершение eval происходит после того, как сообщение об исключении будет записано в переменную $@ ($EVAL_ERROR). Можно попытаться спровоцировать фатальную ошибку посредством оператора die (который возбуждает исключение), но она может быть перехвачена динамически охватывающим eval. В отсутствие перехвата die становится фатальной ошибкой.

фильтр — filter

Программа, предназначенная для преобразования потока ввода в поток вывода.

фильтр ввода/вывода — I/O layer

Один из фильтров между исходными данными и тем, что получится на входе или на выходе.

фильтр исходного кода — source filter

Модуль особого типа, осуществляющий *препроцессинг* нашего сценария перед тем, как он попадет в *лексический анализатор*.

флаг — flag

Мы стараемся избегать этого термина ввиду его многозначности. Он может означать *ключ командной строки*, который сам не принимает аргументов (например, флаги Perl -n и -p), или реже индикаторный разряд (например, флаги O_CREAT и O_EXCL в sysopen).

формальные аргументы – formal arguments

Общие имена, под которыми *подпрограмме* известны ее аргументы. Во многих языках формальным аргументом всегда даются личные имена, но в Perl формальные аргументы просто являются элементами массива. Формальными аргументами в программе Perl являются $ARGV[0], $ARGV[1] и т.д. Аналогично формальными аргументами в подпрограмме Perl являются $_[0], $_[1] и т.д. Аргументам можно дать отдельные имена, присвоив их значения списку my. См. также *фактические аргументы*.

формат – format

Указывает, сколько и куда помещать пробелов, цифр и прочего, чтобы все, что выводится, выглядело аккуратно и красиво.

формирователь – composer

«Конструктор» для *объекта ссылки*, который в действительности не является *объектом*, например, анонимного массива или хеша. Например, пара фигурных скобок действует как формирователь хеша, а пара квадратных скобок действует как формирователь для массива. См. раздел «Создание ссылок» в главе 8.

функция – function

В терминах математики – отображение набора исходных значений в набор конечных значений. В компьютерах указывает на *подпрограмму* или *оператор*, возвращающие значение. Такая функция может и не иметь исходных значений (называемых *аргументами*).

хакер – hacker

Некто, проявляющий поразительную настойчивость в решении технических проблем, будь то игра в гольф, сражение с орками или программирование. Хакер является нейтральным термином в смысле морали. Добрых хакеров не нужно путать со злыми *взломщиками* или невежественными *script kiddies*. Если вы их путаете, мы должны предположить, что вы злы или невежественны.

хеш – hash

Неупорядоченное объединение пар *ключ/значение*, позволяющее использовать строку *ключ* для поиска связанного с ней *значения* данных. Этот глоссарий подобен хешу, в котором определяемое слово служит ключом, а определение является значением. Иногда хеш называют «ассоциативным массивом», и длина этого названия служит достаточным основанием предпочесть слово «хеш».

хеш-таблица – hash table

Структура данных, используемая Perl для эффективной реализации ассоциативных массивов (*хешей*). См. также *блок – bucket*.

хост – host

Компьютер, на котором располагается программа или другие данные.

целое число – integer

Число без дробной (десятичной) части. Числа по порядку, например 1, 2, 3 и т.д., но вместе с числом 0 и отрицательными числами.

цикл – loop

Конструкция, которая делает что-либо многократно, как американские горки.

ЧАВО – FAQ

Frequently Asked Question – часто задаваемые вопросы (не обязательно с часто получаемыми ответами, особенно если ответ есть в Perl FAQ, входящем в стандартную поставку Perl).

числовой контекст – numeric context

Ситуация, в которой окружение (вызывающий код) предполагает, что выражение возвращает число. См. также *контекст* и *строковый контекст*.

читаемый – readable

Применительно к файлу означает, что установлен бит разрешений, позволяющий обращаться к файлу. Применительно к программе означает, что она написана достаточно хорошо, чтобы можно было разобраться в ее устройстве.

члены-данные – member data

См. *переменная экземпляра*.

шаблон – pattern

Структура, используемая при *поиске по шаблону*.

шаблон этапа выполнения – runtime pattern

Шаблон, содержащий одну или более переменных, которые должны быть интерполированы до разбора шаблона как *регулярного выражения*, который поэтому нельзя анализировать на этапе компиляции, но следует заново анализировать при каждом выполнении оператора поиска по шаблону. Шаблоны этапа выполнения полезны, но дороги.

шестнадцатеричный – hexadecimal

Число по основанию 16, «hex». Числа от 10 до 16 обычно представляются буквами от a до f. Шестнадцатеричные константы в Perl начинаются с 0x. См. также функцию hex в главе 27.

широковещание – broadcast

Отправка *датаграммы* сразу по нескольким адресам.

экземпляр – instance

Сокращенно от «экземпляр класса», что означает *объект класса*.

эклектичный – eclectic

Порожденный многими источниками. Некоторые скажут, что *слишком* многими.

эксплоит, уязвимость – exploit

В данном случае это слово используется как существительное. Оно обозначает известный способ скомпрометировать программу и заставить ее выполнить действия, не предусмотренные автором. Ваша задача в том, чтобы писать программы, не допускающие такой возможности.

экспорт, экспортировать – export

Сделать символы *модуля* доступными для *импорта* другими модулями.

элемент – element

Основной строительный элемент. Если говорить о *массиве*, элемент – это один из предметов, из которых состоит массив.

этап выполнения – runtime

Время, когда Perl фактически выполняет то, что сказано в нашем коде, в противоположность предшествующему периоду, *этапу компиляции*, когда Perl пытался выяснить, есть ли вообще смысл в том, что мы сказали.

этап компиляции – compile time

Этап, на котором Perl пытается осмыслить ваш код, в противоположность тому времени (*этап выполнения*), когда ему кажется, что он знает, каков смысл вашего кода, и просто пытается сделать то, что, по его мнению, ваш код хочет, чтобы было сделано.

Юникод – Unicode

Набор символов, включающий практически все значимые наборы символов на свете. См. *http://www.unicode.org*.

является – is-a

Разновидность связи между двумя *объектами*, когда один объект рассматривается как более специфическая версия другого, родового объекта: «верблюд относится к млекопитающим». Поскольку родовой объект в действительности существует только в платоническом смысле, мы обычно добавляем долю абстракции к понятию объектов и представляем связь как существующую между родовым *базовым классом* и специфическим *производным классом*. Как ни странно, у платонических классов связи бывают не только платоническими – см. *наследование*.

Алфавитный указатель

Symbols

C

D

E

N

O

P

Q

R

T

V

W

X

Y

Z

Cyrillic

А

Б

Л

О

П

Р

С

Х

Ц

Ч

Ш

Э

Ю

Я

Издательство "СИМВОЛ-ПЛЮС"

Основано в 1995 году

О нас

Наша специализация – книги компьютерной и деловой тематики. Наши издания – плод сотрудничества известных зарубежных и отечественных авторов, высококлассных переводчиков и компетентных научных редакторов. Среди наших деловых партнеров издательства: O'Reilly, Pearson Education, NewRiders, Addison Wesley, Wiley, McGraw-Hill, No Starch Press, Packt, Dorset House, Apress и другие.





















Apress MANNING

Где купить

Наши книги вы можете купить во всех крупных книжных магазинах России, Украины, Белоруссии и других стран СНГ. Однако по минимальным ценам и оптом они продаются:

Санкт-Петербург:
главный офис издательства –
В.О. 16 линия, д. 7 (м. Василеостровская),
тел. (812) 380-5007

Москва:
московский филиал издательства –
ул. 2-я Магистральная, д. 14В
(м. Полежаевская/Беговая),
тел. (495) 638-5305

Заказ книг

через Интернет http://www.symbol.ru

Бесплатный каталог книг высылается по запросу.

Приглашаем к сотрудничеству



Мы приглашаем к сотрудничеству умных и талантливых авторов, переводчиков и редакторов. За более подробной информацией обращайтесь, пожалуйста, на сайт издательства www.symbol.ru.

Также на нашем сайте вы можете высказать свое мнение и замечания о наших книгах. Ждем ваших писем!